A Why3 proof of GMP algorithms

Raphaël Rieu-Helft

A Why3 proof of GMP algorithms

Raphaël Rieu-Helft
TrustInSoft
Inria

Large-integer arithmetic algorithms are used in contexts where both their performance and their correctness are critical, such as cryptographic software. The fastest algorithms are complex enough that formally verifying them is desirable but challenging. We have formally verified a comprehensive arbitrary-precision integer arithmetic library that implements many state-of-the-art algorithms from the GMP library. The algorithms we have verified include addition, subtraction, Toom-Cook multiplication, division and square root. We use the Why3 platform to perform the proof semi-automatically. We obtain an efficient and formally verified C library of low-level functions on arbitrary-precision natural integers. This verification covers the functional correctness of the algorithms, as well as the memory safety and absence of arithmetic overflows of their C implementation. Using detailed pseudocode, we present the algorithms that we verified and outline their proofs.

1. INTRODUCTION

The GNU Multi-Precision library, or GMP,[1] is a very widely used library for arbitrary-precision arithmetic. It provides state-of-the-art algorithms for basic arithmetic operations on integer, rational and floating-point numbers as well as number-theoretic primitives. It is used in safety-critical contexts such as cryptography and security of Internet applications. For performance reasons, many of these algorithms are very intricate, which makes the presence of correctness bugs a real possibility. Moreover, even proving the absence of crashes and memory issues sometimes relies on functional correctness properties, such as the fact that a carry propagation cannot go past the bounds of an array.

The library is extensively tested, but some parts of the code are visited only in very unlikely cases, such as a $1/2^{64}$ probability (e.g. a case in long division that occurs when two machine words have the same value, assuming uniformly distributed inputs). This makes random testing very challenging. Correctness bugs occuring with very low probability have in fact been found in the past.[2] In order to ensure the absence of correctness bugs, formal verification for all possible inputs is therefore desirable.

GMP has multiple layers, which handle different types of numbers (natural, relative, rational, floating-point) at different levels of abstraction. This work focuses only on the lowest-level layer of GMP, called `mpn`, which provides arithmetic primitives for arbitrary-sized natural integers. We did not verify the entire `mpn` layer. Indeed, for each operation, `mpn` implements many different algorithms, each most suitable for different input sizes. The fragment of `mpn` that we have verified includes at least one algorithm for each of addition, subtraction, multiplication, division and square root (fast modular exponentiation is a work-in-progress). This paper mostly focuses on the functional correctness of the algorithms from a mathematical point of view. Indeed, most implementation-level concerns are

[1] `https://gmplib.org/`
[2] Look for 'division' at `https://gmplib.org/gmp5.0.html`.

automatically proved with very little user input on top of the program code. This leaves the proof of higher-level mathematical facts as the only non-trivial part of the proof. Furthermore, the Why3 mechanized proofs of these mathematical facts are very similar to paper proofs. Therefore, the proofs we present in this work deal with faithful transcriptions of the algorithms in detailed pseudocode. We give the specifications and invariants, and provide step-by-step explanations of the algorithms and proofs that the implementations indeed match the specifications.

Our approach is to implement the GMP algorithms (originally implemented in C and handwritten assembly) in WhyML, the high-level programming language supported by the Why3 verification environment. We give the algorithms a formal specification, and verify that they fulfill these specifications with the help of Why3 and automated theorem provers. Our functions are implemented on top of a Why3 model of the C language that accounts for memory safety and lack of runtime errors. Thus, they can be implemented using the low-level expressivity of C that we need to mirror GMP's own implementation. The model also permits a direct compilation from WhyML to C. Thus, we obtain a formally verified C library that is both compatible with GMP and almost as efficient for inputs smaller than about 1,000 bits (100,000 for multiplication). The full development is available at `http://toccata.lri.fr/gallery/multiprecision.en.html`. We have found no bugs in GMP while performing this verification, but the developers modified a section of a Toom-Cook algorithm (Toom-2) after we pointed out that its correction relied on much more intricate reasonings than what seemed needed. Section 2 provides some context on the process of Why3 proofs. As our Why3 model of C has already been presented [24], the rest of this paper focuses on the higher-level, mathematical aspects of the verification.

There are several challenges to overcome when carrying out such a verification work. The first is to understand why the algorithms are in fact correct. This is easy for the algorithms that we discuss in Section 3 (comparison, addition, subtraction, schoolbook multiplication), but represents a significant part of the work for most algorithms. Some of the optimizations layered over the algorithms obfuscate them somewhat, such as the division algorithm in Section 4. Moreover, seemingly innocuous statements sometimes require a complex proof in order to show that, for example, an operation or a carry propagation does not overflow. Section 5, which focuses on the Toom-Cook divide-and-conquer multiplication, provides good examples of this. Textbooks such as Brent and Zimmermann's [6] explain the broad strokes of most of the algorithms, but do not mention any of these implementation tricks in their pseudocode, not to mention that some algorithms are simply more recent than the textbooks. Many of these algorithms do not seem to have been published by their original authors. The second challenge is to structure the proof in such a way that Why3 and the automated provers can replay it. Due to the nature of the weakest-precondition calculus used by Why3, all loops must be annotated with inductive invariants, and all functions must carry extensive specifications. Furthermore, the automated provers we use do not handle logical goals that involve non-linear arithmetic and numbers with unknown size very well. In practice, the user sometimes has to provide extensive proof hints in order to prove seemingly simple facts.

We discuss GMP's square root algorithm in Section 6. The divide-and-conquer part of it has already been verified in Coq by Bertot et al. [3], and our proof is largely adapted from theirs. Related work, covered in Section 7, otherwise mostly comprises formal proofs of single state-of-the-art algorithms, and verified libraries of simpler, less efficient algorithms. To the best of our knowledge, this work is the first formal verification of a comprehensive

state-of-the-art arbitrary-precision integer library.

2. WHY3 AND GMP

The goal of this section is to provide some context on the process of verifying GMP algorithms. We will first go over GMP's number representation, and then explain the process of Why3 proofs, using a very simple GMP routine as an example.

2.1 GMP number representation

In GMP, natural integers are represented as arrays of unsigned integers called *limbs*. We set a radix $\beta = 2^{64}$ (also called `radix` in the formal development). Any natural number N has a unique decomposition $\sum_{k=0}^{n-1} a[k]\beta^k$ in base β such that $a[n-1] \neq 0$, and is represented as the buffer $a[0]a[1]\dots a[n-1]$ (with the least significant limb first).

For efficiency, there is no memory management in GMP's low-level functions, so the caller code has to keep track of number sizes. Instead, operands are specified by a pointer to their least significant limb and a limb count of type `int32`.

```
type limb = uint64

type t = ptr limb
```

A pointer is considered *valid* over a size s if it is not null and points to a zone of size at least s.

Let us now establish the link between mathematical integers and arrays of machine integers. If a pointer a is valid over a size n, we denote $\texttt{value}(a, n) = \overline{a[0]\dots a[n-1]} = \sum_{k=0}^{n-1} a[k]\beta^k$. We also denote $\texttt{value}(\texttt{incr}(a, k), p)$ as $\overline{a[k]...a[k+p-1]}$.

We use a few lemmas to express what happens to the value of a big integer when part of it is modified. In particular, loops tend to change only one end of a big integer (usually by increasing its length), and being able to separate what changed from what did not is crucial to prove that loop invariants are preserved.

LEMMA 1. *Let p a pointer valid over a sufficiently large length.*

$$\forall k, \overline{p[0]\dots p[n-1]} = \overline{p[0]\dots p[k-1]} + \beta^k \overline{p[k]\dots p[n-1]} \qquad \texttt{[value\_sub\_concat]}$$

$$\overline{p[0]\dots p[n]} = \overline{p[0]\dots p[n-1]} + \beta^n p[n] \qquad \texttt{[value\_sub\_tail]}$$

$$\overline{p[0]\dots p[n-1]} = p[0] + \beta\overline{p[1]\dots p[n-1]} \qquad \texttt{[value\_sub\_head]}$$

$$\forall k, v, \overline{p[0]\dots p[k-1]vp[k+1]\dots p[n-1]}$$
$$= \overline{p[0]\dots p[k]\dots p[n-1]} + \beta^k(v - p[k]) \qquad \texttt{[value\_sub\_update]}$$

We also need some bounds for the value of an integer of size n. These follow easily from the fact that the values $a[i]$ lie between 0 and β.

LEMMA 2. *Let p a pointer valid over the size n.*

$$0 \leq \overline{p[0]\dots p[n-1]} \qquad \texttt{[value\_sub\_lower\_bound]}$$

$$p[n-1]\beta^{n-1} \leq \overline{p[0]\dots p[n-1]} \qquad \texttt{[value\_sub\_lower\_bound\_tight]}$$

$$\overline{p[0]\dots p[n-1]} < \beta^n \qquad \texttt{[value\_sub\_upper\_bound]}$$

$$\overline{p[0]\dots p[n-1]} < (p[n-1]+1)\beta^{n-1} \qquad \texttt{[value\_sub\_upper\_bound\_tight]}$$

2.2 From program to logical goals

Let us now go over the process of Why3 proofs with the example of GMP's copy routine, shown in Alg. 1. It takes two numbers r and x valid over a size n and copies the contents of x into r, starting with the least significant limbs. Note that x and r are allowed to point to zones longer than n, or even to point to the middle of a number. The only requirement is that there are at least n valid limbs to the right of each pointer.

Algorithm 1 Copy of a long integer

function COPYI(r, x, n)
  $i \leftarrow 0$
  while $i < n$ **do**
    $r[i] \leftarrow x[i]$
    $i \leftarrow i + 1$

The Why3 deductive program verification platform provides a language for specification and programming called WhyML, the programming part of which is a dialect of ML. The first step of a Why3 proof is to translate the program that we want to prove in WhyML. We also use WhyML to write a contract for the program. For the COPYI function, the (slightly simplified) Why3 implementation is shown in Figure 1.

```
let wmpn_copyi (r x: ptr uint64) (n: int32) : unit
=
  let ref i = 0 in
  while (Int32.(<) i n) do
    r[i] ← x[i];
    i ← i+1;
  done
```

Fig. 1. WhyML transcription of the COPYI function.

WhyML functions are annotated with contracts based on Hoare logic [17, 15]. Given an expression e, a precondition P and a postcondition Q, where P and Q are assertions in first-order logic, one can write the *Hoare triple* $\{P\}e\{Q\}$. Its intuitive reading is: Whenever the state before the execution of e is such that P holds, then the computation terminates, there is no runtime error, and the final state satisfies Q. Hoare triples can be derived from a set of inference rules based on program syntax. A soundness theorem implies that any derivable triple is correct. Some examples are shown in Figure 2.

$$\frac{}{\{P\}\texttt{skip}\{P\}} \qquad \frac{}{\{P[x \leftarrow e]\}x \leftarrow e\{P\}} \qquad \frac{\{P\}s_1\{Q\} \quad \{Q\}s_2\{R\}}{\{P\}s_1; s_2\{R\}}$$

Fig. 2. Some Hoare triple inference rules.

The contract of a function forms a Hoare triple with the function body. The precondition P is the conjunction of the `requires` clauses, and the postcondition Q is the conjunction of the `ensures` clauses. For example, the contract of COPYI is as follows:

```
let wmpn_copyi (r x: ptr uint64) (n: int32) : unit
  requires { valid x n ∧ valid r n }
  ensures  { forall i. 0 ≤ i < n → r[i] = x[i] }
  ensures  { forall i. i < 0 ∨ n ≤ i → r[i] = old r[i] }
```

Using this contract, Why3 produces a logical goal that implies that the program satisfies this specification. The aim is to find out whether the Hoare triple defined by the contract is valid. Using the inference rules naively would be far too tedious. Indeed, the rule for sequences of instructions (see Figure 2) requires an intermediate assertion between each pair of statements. The code would need to be very heavily annotated for the naive approach to work.

Instead, Why3 uses Dijkstra's weakest-precondition calculus [10]. We define the predicate transformer $\mathrm{WP}(.,.)$. Where e is an expression and Q a postcondition, $\mathrm{WP}(e, Q)$ computes the weakest precondition P such that $\{P\}e\{Q\}$ holds. This is a way to automate the search for intermediate assertions in the derivations. A subset of the computation rules for $\mathrm{WP}(.,.)$ can be found in Figure 3.

$$\begin{aligned}\mathrm{WP}(\texttt{skip}, Q) &= Q\\ \mathrm{WP}(e_1; e_2, Q) &= \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q))\\ \mathrm{WP}(\texttt{assert } R, Q) &= R \wedge (R \to Q)\end{aligned}$$

Fig. 3. Some WP rules.

The soundness theorem of WP states that for any e and Q, the triple $\{\mathrm{WP}(e, Q)\}e\{Q\}$ is valid. Therefore, to show that a contract $\{P\}e\{Q\}$ is valid, it suffices to prove that $P \to \mathrm{WP}(e, Q)$. Why3 computes $\mathrm{WP}(e, Q)$ and outputs $P \to \mathrm{WP}(e, Q)$ as a logical goal for the user to prove.

One major difficulty remains. The previous figures do not show the rules for deriving Hoare triples or computing WP for loops. This problem is undecidable in general, as the loop would need to be unrolled until termination. Therefore, WhyML loops must be annotated with an inductive loop invariant. The corresponding WP computation makes sure that this invariant is valid at the start of the loop, maintained through one iteration of the loop, and that at the end of the loop, it implies the required postcondition. A simplified version of this rule is shown in Figure 4. It assumes that the only modification of the memory state in the loop occurs on a variable v, and does not handle termination.

$$\begin{aligned}&\mathrm{WP}(\texttt{while } t \texttt{ invariant } J \texttt{ do } e \texttt{ done}, Q) \equiv\\ &\quad J \wedge && \text{[invariant initialisation]}\\ &\quad \forall v.\\ &\qquad (J \wedge t \to \mathrm{WP}(e, J)) \wedge && \text{[invariant preservation]}\\ &\qquad (J \wedge \neg t \to Q) && \text{[loop exit and postcondition]}\end{aligned}$$

Fig. 4. Simplified WP rule for `while`.

The only problem that remains is proving termination, which would also be undecidable in general. Why3 requires that proofs and recursive functions be annotated with a variant.

This variant is a program expression that is decreasing at each loop iteration (or function call) on a well-founded order. In the end, the annotated version of COPYI is as follows:

```
let wmpn_copyi (r x: ptr uint64) (n: int32) : unit
  requires { valid x n ∧ valid r n }
  ensures  { forall i. 0 ≤ i < n → r[i] = x[i] }
  ensures  { forall i. i < 0 ∨ n ≤ i → r[i] = old r[i] }
= let ref i = 0 in
  while (Int32.(<) i n) do
    variant { n - i }
    invariant { forall j. 0 ≤ j < i → r[j] = x[j] }
    invariant { forall j. j < 0 ∨ i ≤ j → r[j] = old r[j] }
    r[i] ← x[i];
    i ← i+1;
  done
```

Note that it is not necessary to specify that the memory block pointed by `x` was not modified, which would happen if `x` and `r` were aliased. Indeed, Why3's region-based type system imposes strong constraints upon WhyML programs, in such a way that all aliases are known statically and encoded in the types [16]. We used this fact to design our C memory model in such a way that pointers used as function parameters are forced to be separated by typing (unless they are read-only). Moreover, typing enforces that separated pointers cannot reach each other through pointer arithmetic. This makes proofs much simpler, both in terms of specification length (many more annotations would otherwise be necessary) and in terms of size of the proof context.

2.3 Proving the goals

Once Why3 has generated the logical goals that imply the program meets its specification, they should be proved by the user. Why3 offers several ways for them to do so. The most important one is to dispatch the proof obligations to a variety of automated theorem provers. Why3 has drivers for various provers, such as SMT solvers Alt-Ergo, Z3, and CVC4, as well as superposition-based provers such as E, SPASS and Vampire. The default way of proving a goal is to send it to one or more SMT solvers. If at least one of them returns that the goal is correct, Why3 considers it proved. Why3 offers task transformations that can be used, for example, to split a very large goal such as the weakest-precondition of a large program into many tractable ones.

While SMT solvers are very powerful tools in many contexts, sometimes proof obligations fall outside their scope and they do not manage to prove them in any reasonable timeframe. For example, the theory of nonlinear integer arithmetic is undecidable. While solvers do manage to solve some nonlinear arithmetic goals using various proof search techniques, many goals that naturally arise during the verification of GMP algorithms are not solved by any of the provers that we tried. Furthermore, the fact that GMP integers are arbitrary-sized and the large size of the proof contexts make these goals even harder to solve using non-complete heuristics.

When simply sending the goal to automated solvers is not enough to prove it in a reasonable time, Why3 offers several tools for the user to provide extra guidance. First, the user can write a proof in an interactive theorem prover, such as Coq, Isabelle or PVS. This approach is generally quite tedious, and we used it only for a handful of goals in this work. Second, the Why3 user can declare and prove lemmas. These lemmas are passed to the solvers alongside the rest of the proof context. The user may also call them like functions

in the code, in order to force their application on specific arguments. Figure 5 shows an example of this. The first lemma is declared and proved trivially, and the second lemma is proved using the first lemma twice. Note that arbitrary control structures can be used in the proof of lemmas, as they are simply ghost WhyML functions. For example, the third lemma is proved by induction over `n`, using an `if` statement to split between the base case and the inductive case. The right column shows the interpretation of these functions as lemmas in Why3's logical context.

```
let lemma prod_compat_r (a b c:int)
  requires { 0 ≤ a ≤ b }
  requires { 0 ≤ c }
  ensures { c * a ≤ c * b }
= ()
```

$$\forall a, b, c \in \mathbb{Z}, 0 \leq a \leq b \wedge 0 \leq c \implies ca \leq cb$$

```
let lemma prod_compat_lr (a b c d:int)
  requires { 0 ≤ a ≤ b }
  requires { 0 ≤ c ≤ d }
  ensures  { a * c ≤ b * d }
= prod_compat_r c d a;  (* ac ≤ ad *)
  prod_compat_r a b d   (* da ≤ db *)
```

$$\forall a, b, c, d \in \mathbb{Z}, 0 \leq a \leq b \wedge 0 \leq c \leq d \implies ac \leq bd$$

```
let rec lemma pow2_gt (n:int)
  requires { 0 ≤ n }
  ensures { n ≤ power 2 n }
  variant { n }
= if n > 0 then pow2_gt (n-1)
```

$$\forall n \in \mathbb{Z}, 0 \leq n \implies n \leq 2^n$$

Fig. 5. Declaring, proving and using lemmas.

A third way to prove difficult goals is to write assertions inside the code. The assertions are logical formulas that are interpreted as proof cuts, using the WP rules for sequences and `assert` in Figure 3. Why3 checks that the assertion is correct using the same methods as for the other goals, and then inserts it in the logical context. Figure 6 features a Why3 lemma and its proof, which consists in a large assertion. In our case, the assertion is a full-fledged proof, written using the logical connectives `by` and `so` [7]. The proof is essentially a sequence of implications, which Why3 checks using automated provers. The end result is essentially what we would expect from a paper proof. Much like in a paper proof, when developing such a Why3 assertion, we simply add extra steps when the automated provers do not manage to check an implication. Many of the mathematical proofs that are given in this paper were proved using this method in our Why3 development.

Finally, another way for the Why3 user to prove difficult goals is computational reflection. Why3 allows a user to define and verify dedicated decision procedures as WhyML programs, and to execute these procedures in order to prove the goals [20]. We used this approach in our development to prove goals that previously required hundreds of lines of handwritten assertions.

Using these various tools, it is usually not too difficult to go from a detailed paper proof to a Why3 mechanized proof. Aside from a few goals related to the absence of overflow in some operations (which we do discuss), most implementation-level concerns are automatically discharged by the SMT solvers with next to no work on our part. The proof effort

```
let lemma fact_div (x y z:int)
  requires { y > 0 }
  ensures  { div (x + y * z) y = (div x y) + z }
=
  assert { div (x + y * z) y = (div x y) + z
           by x + y * z = y * (div (x + y * z) y) + mod (x + y * z) y
           so mod (x + y * z) y = mod (y * z + x) y = mod x y
           so x + y * z = y * (div (x + y * z) y) + mod x y
           so x = y * div x y + mod x y
           so x + y * z = y * div x y + mod x y + y * z
           so y * (div (x + y * z) y) + mod x y
              = y * div x y + mod x y + y * z
           so y * (div (x + y * z) y)
              = y * div x y + y * z
              = y * ((div x y) + z)
           so div (x + y * z) y = div x y + z }
```

Fig. 6. A Why3 proof using a large assertion.

was overwhelmingly concentrated on the mathematical correctness of the algorithms. In the rest of this paper, we will therefore focus on proving the functional correctness of the various algorithms on paper, while abstracting some language-specific concerns away. Furthermore, the Why3 proofs of mathematical correctness are extremely similar to paper proofs, as exemplified by the proof in Figure 6. Therefore, we will focus on proving the correctness of the algorithms on paper, rather than showing the Why3 translation of these paper proofs.

3. BASIC ALGORITHMS

3.1 Comparisons

The `mpn` layer of GMP exposes a single comparison function, which compares two integers of same length (Alg. 2). The algorithm is very straightforward: it simply iterates both operands until it finds a difference, starting at the most significant limb.

Algorithm 2 Comparison of two integers of identical length.

function CMP(x, y, n)
Require: $\texttt{valid}(x, n), \texttt{valid}(y, n)$
Ensure: $\texttt{result} > 0 \Leftrightarrow \texttt{value}(x, n) > \texttt{value}(y, n)$
Ensure: $\texttt{result} = 0 \Leftrightarrow \texttt{value}(x, n) = \texttt{value}(y, n)$
Ensure: $\texttt{result} < 0 \Leftrightarrow \texttt{value}(x, n) < \texttt{value}(y, n)$
    for $i = n - 1$ **downto** 0 **do**
        if $x[i] \neq y[i]$ **then**
            if $x[i] > y[i]$ **then return** 1
            else return -1
    return 0

Since the function involves a loop, we must provide a loop invariant. Here, the loop invariant is that both source operands are identical from offsets $i + 1$ to n.

An additional lemma is required to prove the invariant and complete the proof, it simply says that two big integers with equal limbs at all offsets are equal:

LEMMA `value_sub_frame`.
Let $a[0], \ldots, a[n-1], b[0], \ldots, b[m-1]$ *such that for all* $u \leq k \leq v$, $a[k] = b[k]$. *Then* $a[u] \ldots a[v] = b[u] \ldots b[v]$.

The lemma is proved by a straightforward induction, which translates well into a Why3 lemma function as the recursive call takes care of the inductive case:

```
let rec lemma value_sub_frame (a b:map int limb) (u v:int)
  requires { forall i. u ≤ i < v → a[i] = b[i] }
  variant  { m - n }
  ensures  { value_sub a u v = value_sub b u v }
= if u < v then value_sub_frame a b (u+1) v else ()
```

This lemma shows that the numbers are equal if no difference was found by the end of the loop.

3.2 Addition, Subtraction

We use the schoolbook algorithms for the addition and subtraction of big integers, represented by their decomposition in base β.

We first need to give the specifications of basic operations on limbs. The following three primitives can be used to add limbs.

The first primitive + is the defensive addition: it requires that the sum of the two inputs does not overflow.

```
val (+) (a b:limb) : limb
  requires { "expl:integer overflow" Limb.min ≤ to_int a + to_int b ≤ Limb.max }
  ensures  { to_int result = to_int a + to_int b }
```

The second primitive, `add_mod`, has the semantics of the + operator on unsigned integers in C: if there is an overflow, the result wraps around. When compiling WhyML programs to C, both `(+)` and `add_mod` are translated to the C operator +. The former has a stronger postcondition, so using it simplifies the proofs when its (also stronger) precondition is met. The latter captures the full semantics of the addition, so it can be used in the remaining cases.

```
val add_mod (x y:limb) : limb
  ensures { to_int result = mod (to_int x + to_int y) radix }
```

Finally, the third primitive accepts a carry to be added to the other two operands, and outputs both the carry and the (potentially wrapped-around) result of the addition.

```
val add_with_carry (x y:limb) (c:limb) : (limb,limb)
  requires { 0 ≤ to_int c ≤ 1 }
  returns { (r,d) →
    to_int r + radix * to_int d =
    to_int x + to_int y + to_int c
    ∧ 0 ≤ to_int d ≤ 1 }
```

Similar primitives are used for limb subtraction.

Our library implements many variants of addition and subtraction (depending on whether the operation is done in place, the operation may overflow, the operands are known to be of same length, etc.) As an example, let us examine the general-case addition (Alg. 3).

Algorithm 3 Addition of two integers

Require: $0 \leq n \leq m$
Require: $\texttt{valid}(a, m), \texttt{valid}(b, n), \texttt{valid}(r, m)$
Ensure: $\texttt{value}(r, m) + \beta^m \cdot \texttt{result} = \texttt{value}(a, m) + \texttt{value}(b, n)$
Ensure: $0 \leq \texttt{result} \leq 1$
function ADD(r, a, m, b, n)
  $i \leftarrow 0$
  while $i < n$ **do** ▷ Add b to a.
    $x \leftarrow a[i]$
    $y \leftarrow b[i]$
    $(z, c) \leftarrow$ ADD_WITH_CARRY(x, y, c)
    $r[i] \leftarrow z$
    $i \leftarrow i + 1$
  if $c \neq 0$ **then** ▷ Keep copying a into r while propagating the carry.
    while $i < m$ **do**
      $x \leftarrow a[i]$
      $z \leftarrow$ ADD_MOD$(x, 1)$ ▷ $c = 1$.
      $r[i] \leftarrow z$
      $i \leftarrow i + 1$
      if $z \neq 0$ **then** ▷ No overflow: there is no more carry.
        $c \leftarrow 0$
        break
  while $i < m$ **do** ▷ No more carry: copy a into r.
    $r[i] \leftarrow a[i]$
    $i \leftarrow i + 1$
  return c

The algorithm is schoolbook addition with a few optimisations. There are three main steps.

The first step is to add together the two operands over the length of the shorter one. This corresponds to the first `while` loop. Its loop invariants are as follows:

(1) $\texttt{value}(r, i) + c\beta^i = \texttt{value}(a, i) + \texttt{value}(b, i)$
(2) $0 \leq i \leq n$.

At the end of the first loop, we have $\texttt{value}(r, n) + c\beta^n = \texttt{value}(a, n) + \texttt{value}(b, n)$ and $i = n$. What remains to be done is to copy the last $m - n$ limbs of a into r and propagate the carry.

The second loop copies a into r while propagating the carry. It is skipped if $c = 0$. Its loop invariants are:

(1) $\texttt{value}(r, i) + c\beta^i = \texttt{value}(a, i) + \texttt{value}(b, n)$,
(2) $n \leq i \leq m$,
(3) $i = m \vee c = 1$.

We break out of the loop whenever the carry c becomes 0 (or if we have finished copying a into r, in which case $i = m$). Note that we are only adding the carry to a limb x of a. There is an overflow if and only if $x = \beta - 1$. There is no need to use ADD_WITH_CARRY. It is more efficient to instead use ADD_MOD and check if the result is 0, in which case there

was an overflow. This is relatively unlikely (probability $1/\beta$ if the operands are randomly drawn from a uniform distribution), so the loop typically only runs for zero or one iteration.

Finally, in the third loop (which is skipped if we already have $i = m$), we only have to copy the last limbs of a into r. The loop invariants are:

(1) $\texttt{value}(r, i) + c\beta^i = \texttt{value}(a, i) + \texttt{value}(b, n)$,
(2) $n \leq i \leq m$,
(3) $i = m \vee c = 0$.

At the end of the loop, we can return c and easily see that the postcondition is verified.

3.3 Schoolbook multiplication

We have implemented several algorithms for integer multiplication. For smaller integers (fewer than 30 limbs), the fastest is the schoolbook one, which has complexity $O(n^2)$. For larger operands, Toom-Cook multiplication is used (Section 5), as it has better asymptotic complexity.

Algorithm 4 Multiply-and-add

Require: $\texttt{valid}(r, m), \texttt{valid}(a, m)$
Ensure: $\texttt{value}(r, m) + \beta^m \cdot \texttt{result} = \texttt{value}(\texttt{old } r, m) + \texttt{value}(a, m) \times y$
Ensure: $\forall j.j < 0 \vee m \leq j \Rightarrow r[j] = (\texttt{old } r)[j]$
function ADDMUL_1(r, a, m, y)
  $c, i = 0$
  while $i < m$ **do**
    $x \leftarrow a[i]$
    $z \leftarrow r[i]$
    $(l, h) \leftarrow$ MUL_DOUBLE(x, y) ▷ $l + \beta \cdot h = x \cdot y$
    $v, c' \leftarrow$ ADD3(z, l, c) ▷ $v + \beta \cdot c' = z + l + c$
    $r[i] \leftarrow v$
    $c \leftarrow c' + h$
    $i \leftarrow i + 1$
  return c

Let us first consider the auxiliary function ADDMUL_1 (Alg. 4). It multiplies a big integer a by a limb y and adds the result to r, without modifying the contents of r outside the area of the addition.

The loop invariants of ADDMUL_1 are:

(1) $0 \leq i \leq m$,
(2) $\texttt{value}(r, i) + c\beta^i = \texttt{value}(\texttt{old } r, i) + \texttt{value}(a, i) \cdot y$,
(3) $\forall j.j < 0 \vee m \leq j \Rightarrow r[j] = (\texttt{old } r)[j]$.

Let us now consider the main function that implements schoolbook multiplication (Alg. 5). At each loop iteration, one limb of the second operand is multiplied by the entire first operand, and the product is added to the result (shifted appropriately).

The loop invariants are as follows:

(1) $1 \leq i \leq n$,
(2) $\texttt{value}(r, m + i) = \texttt{value}(a, m) \cdot \texttt{value}(b, i)$,

Algorithm 5 Schoolbook multiplication of two integers

Require: $0 < n \leq m$
Require: $\texttt{valid}(a, m), \texttt{valid}(b, n), \texttt{valid}(r, m + n)$
Ensure: $\texttt{value}(r, m + n) = \texttt{value}(a, m) \cdot \texttt{value}(b, n)$
  function MUL_BASECASE(r, a, b, m, n)
    $y \leftarrow b[0]$
    $r[m] \leftarrow \text{MUL\_1}(r, a, y, m)$ ▷ $\texttt{value}(r, m + 1) = \texttt{value}(a, m) \cdot b[0]$.
    $p \leftarrow r + 1$
    $i \leftarrow 1$
    while $i < n$ **do**
      $y \leftarrow b[i]$
      $p[m] \leftarrow \text{ADDMUL\_1}(p, a, m, y)$ ▷ See Alg. 4.
      $i \leftarrow i + 1$
      $p \leftarrow p + 1$

(3) $p = r + i$.

It is easy to see that the postcondition follows from the invariants. The fact that the invariants are maintained follows from the specification of the auxiliary function ADDMUL_1.

Indeed, if we pose r' the state of r at the beginning of the loop, we have at the end of the loop:

$$
\begin{aligned}
&\texttt{value}(r, m + i + 1) \\
&= \texttt{value}(r, i) + \beta^i \cdot \texttt{value}(r + i, m + 1) && \text{decomposition} \\
&= \texttt{value}(r', i) + \beta^i \cdot \texttt{value}(r + i, m + 1) && \text{no writes in } r(0, i) \\
&= \texttt{value}(r', i) + \beta^i \cdot (\texttt{value}(r' + i, m) + \texttt{value}(a, m) \cdot y) && \text{postcondition of ADDMUL\_1} \\
&= \texttt{value}(r', m + i) + \beta^i \cdot \texttt{value}(a, m) \cdot y && \text{recomposition} \\
&= \texttt{value}(a, m) \cdot \texttt{value}(b, i) + \beta^i \cdot \texttt{value}(a, m) \cdot y && \text{loop invariant} \\
&= \texttt{value}(a, m) \cdot (\texttt{value}(b, i) + \beta^i \cdot y) \\
&= \texttt{value}(a, m) \cdot \texttt{value}(b, i + 1). && \text{recomposition}
\end{aligned}
$$

4. DIVISION

Long division consists in computing the quotient and remainder of the division of big integers of arbitrary sizes. It is a significantly more complex problem than long addition and multiplication. While GMP's algorithm is a variation on the schoolbook algorithm, it is thoroughly optimized to the point of making it hard to understand and prove. The GMP general-case long division function is about 30-line long,[3] and the code extracted from our implementation has about the same length. However, our proof for it is about 2000-line long.

Let us first review a more naïve algorithm: Knuth's Algorithm D [19, p.257] (see also [28]), shown in Alg. 6. We did not use this algorithm in our development, but it is simple enough to explain the core ideas more easily.

[3] `mpn/generic/sbpi1_div_qr.c` in GMP 6.1.2

Algorithm 6 Knuth's Algorithm D

Require: $m \geq n > 0, \texttt{valid}(a, m), \texttt{valid}(d, n), \texttt{valid}(q, m-n), \texttt{valid}(r, n)$
Require: $d[n-1] \geq \beta/2$
Require: $\texttt{value}(a+m-n, n) < \texttt{value}(d, n)$ ▷ Otherwise an extra quotient limb is needed.
Ensure: $\texttt{value}(a, m) = \texttt{value}(d, n) \times \texttt{value}(q, m-n) + \texttt{value}(r, n)$
Ensure: $\texttt{value}(r, n) < \texttt{value}(d, n)$

1: **function** ALGORITHMD(q, r, a, d, m, n)
2: **for** $j = m-n-1$ **downto** 0 **do**
3: $\hat{q} \leftarrow$ DIV_2BY1$(a[j+n-1], a[j+n], d[n-1])$ ▷ Candidate quotient limb.
4: $\hat{r} \leftarrow \beta a[j+n] + a[j+n-1] - d[n-1] \times \hat{q}$ ▷ Candidate remainder.
`adjust:`
5: **if** $\hat{q} \geq \beta$ **or** $\hat{q} \times d[n-2] > \beta \times \hat{r} + a[j+n-2]$ **then**
6: $\hat{q} \leftarrow \hat{q} - 1$ ▷ Quotient is too large; adjust.
7: $\hat{r} \leftarrow \hat{r} + d[n-1]$
8: **if** $\hat{r} < \beta$ **then goto** `adjust` ▷ Happens at most once.
9: $b \leftarrow$ SUBMUL_IN_PLACE$(a+j, d, \hat{q}, n)$ ▷ Subtract $\overline{d} \times \hat{q}$ from a.
10: $q[j] \leftarrow \hat{q}$
11: **if** $b > 0$ **then** ▷ There was a borrow, the quotient was too large.
12: $q[j] \leftarrow q[j] - 1$
13: $c \leftarrow$ ADD_N_IN_PLACE$(a+j, d, n)$
14: $a[j+n] \leftarrow a[j+n] + c$ ▷ Propagate the carry.
15: **for** $i = 0$ **to** $n-1$ **do** ▷ The remainder is written in a, copy it to r.
16: $r[i] \leftarrow a[i]$
return

We assume a primitive DIV_2BY1 that divides a 2-limb integer by a 1-limb integer and returns the quotient. It has no WhyML code and we assume that the hardware provides such a function. It is the only division primitive used by the functions in this section.

```
val div_2by1 (l h d:limb) : limb
  requires { to_int h < to_int d }
  ensures { to_int result = div (to_int l + (max_uint64+1) * to_int h) (to_int d) }
```

The algorithm consists in computing the limbs of the quotient one by one, starting with the most significant. The numerator is overwritten at each step to contain the partial remainder.

At each iteration of the loop, we compute a quotient limb and subtract from the current remainder the product of that quotient limb and the denominator, left-shifted appropriately to cancel out the most significant limb of the current remainder.

To compute a quotient limb, a candidate value is first guessed by dividing the two most significant limbs from the current remainder by the most significant limb of the denominator.

This candidate value is then adjusted to match the correct value of the quotient (lines 5-8 and 11-14). This process is called "adjustment step" throughout this section. The algorithms that are actually implemented in GMP are variants of Algorithm D that try to minimize the number of adjustments that occur.

This is where the requirement that $d[n-1] \geq \beta/2$ comes into play. When this is the case (we call such a denominator *normalized*) then the initial 2-by-1 division gives a good approximation of the target quotient limb.

DEFINITION 1. *An integer* $\overline{p[0] \dots p[n-1]}$ *is said to be* normalized *when* $p[n-1] \geq \beta/2$.

```
predicate normalized (x:ptr limb) (sz:int32) =
  valid x sz ∧ x[x.offset + sz - 1] ≥ div radix 2
```

More precisely, as shown by Knuth [19, p.257, Theorem B], the candidate quotient is at most too large by 2, under the condition that the denominator is normalized. The denominator being normalized is therefore a precondition of Knuth's algorithm, and of the other division algorithms in this paper for similar reasons.

In the general case, we remark that an integer is normalized if and only if its most significant bit is set to 1. The denominator can therefore be normalized by counting the leading zeros in the denominator, shifting the numerator and denominator by that amount (the denominator is normalized), calling a division procedure, and correcting the output by shifting the remainder to the right by the same amount.

This normalization is done by a wrapper around the main division primitive. This wrapper is the function that is exposed to the user. We verified a version of the wrapper that is very simple and only performs this normalization, so we will not discuss it in this paper. GMP's version also implements an alternative algorithm that is not needed for correctness, but that is faster than the default one when the divisor is close in length to the dividend. We have not implemented and verified this alternate algorithm yet; the more general algorithm is used in all cases. In the rest of this paper, we will continue to assume the divisor normalized.

4.1 General case algorithm

GMP does not use Knuth's algorithm, but a similar one that uses a 3-by-2 division to compute each quotient limb (Alg. 7). Let us now discuss the differences between this algorithm and Algorithm D.

First, there is an extra local variable x. It is really a proxy for the most significant limb in the current remainder, in the sense that whenever we would read from $a[n+i]$, we take x instead. Thus, instead of being stored in $\overline{a[i] \dots a[n+i]}$, the current remainder is stored in $\overline{a[i] \dots a[n+i-1]}$ with the most significant limb stored separately in x. This saves a few memory accesses, and it can be done because the most significant limb of the current remainder is no longer needed after the current loop iteration.

We are now better equipped to express the main loop invariants.

Let X be equal to the initial value of $\mathtt{value}(a, m)$. The following invariants are maintained in the main loop:

$$X = \mathtt{value}(d, n) \cdot \beta^i \cdot \mathtt{value}(q+i, m-n-i) + \mathtt{value}(a, n+i-1) + \beta^{n+i-1} \times x$$

$$\mathtt{value}(a+i, n-1) + \beta^{n-1} \cdot x < \mathtt{value}(d, n)$$

$$d[n-2] + \beta d[n-1] \geq a[n+i-2] + \beta x \qquad \text{(implied by the previous two).}$$

An important difference with Algorithm D is that instead of dividing the 2 most significant limbs of the numerator by the most significant limb of the denominator, we divide the 3 most significant limbs of the former by the 2 most significant limbs of the latter.

Algorithm 7 General case long division

Require: $m \geq n \geq 3, \texttt{valid}(a, m), \texttt{valid}(d, n), \texttt{valid}(q, m-n)$
Require: $d[n-1] \geq \beta/2$
Require: $\texttt{value}(a+m-n, n) < \texttt{value}(d, n)$ ▷ Otherwise an extra quotient limb is needed
Ensure: $\texttt{value}(\texttt{old } a, m) = \texttt{value}(q, m-n) \times \texttt{value}(d, n) + \texttt{value}(a, n)$
Ensure: $\texttt{value}(a, n) < \texttt{value}(d, n)$

1: **function** DIV_SB_QR(q, a, d, m, n)
2: $v \leftarrow$ RECIPROCAL_WORD_3BY2$(d[n-1], d[n-2])$
3: $x \leftarrow a[m-1]$
4: $i = m - n$
5: **while** $i > 0$ **do**
6: $i \leftarrow i - 1$
7: **if** $x = d[n-1]$ **and** $a[n+i-1] = d[n-2]$ **then** ▷ Unlikely.
8: $\hat{q} \leftarrow \beta - 1$
9: SUBMUL_1$(a+i, d, n, \hat{q})$ ▷ We know the result is $d[n-1]$.
10: $x \leftarrow a[n+i-1]$
11: **else**
12: $(\hat{q}, x, l) \leftarrow$ DIV3BY2_INV$(x, a[n+i-1], a[n+i-2], d[n-1], d[n-2], v)$
13: $b \leftarrow$ SUBMUL_1$(a+i, d, n-2, \hat{q})$
14: $b_1 \leftarrow (l < b)$ ▷ Last two steps of the subtraction are inlined.
15: $a[i+n-2] \leftarrow l - b \mod \beta$
16: $b_2 \leftarrow (x < b_1)$
17: $x \leftarrow x - b_1 \mod \beta$ ▷ Finish subtraction.
18: **if** $b_2 \neq 0$ **then** ▷ Unlikely, and $b_2 = 1$.
19: $\hat{q} \leftarrow \hat{q} - 1$ ▷ We only need to adjust by 1.
20: $c \leftarrow$ ADD_IN_PLACE$(a+i, d, n-1)$ ▷ Add only over $n-1$ limbs.
21: $x \leftarrow x + d[n-1] + c \mod \beta$ ▷ The carry out is always 1.
22: $q[i] \leftarrow \hat{q}$
23: $a[n-1] \leftarrow x$

This means that the adjustment step is much more efficient than that of Algorithm D. Indeed, the candidate quotient obtained this way is very likely to be correct (Lemma 4).

LEMMA 4. *Let $\hat{q}$ and $\overline{r[0]r[1]}$ the quotient and remainder of the division of $\overline{a[n+i-2]a[n+i-1]x}$ by $\overline{d[n-2]d[n-1]}$.*
If $\hat{q} \times \overline{d[0]\dots d[n-1]} > \overline{a[i]\dots a[n+i-1]x}$, then $r[1] = 0$.

PROOF. We have $\overline{a[n+i-2]a[n+i-1]x} = \overline{d[n-2]d[n-1]}\hat{q} + \overline{r[0]r[1]}$.
We also have $\overline{a[i]\dots a[n+i-1]x} \geq \beta^{n-2}\overline{a[n+i-2]a[n+i-1]x}$.
If $\hat{q} \times \overline{d[0]\dots d[n-1]} > \overline{a[i]\dots a[n+i-1]x}$, this implies:

$$\hat{q} \times \overline{d[0]\dots d[n-1]} > \beta^{n-2}(\overline{d[n-2]d[n-1]}\hat{q} + \overline{r[0]r[1]})$$

However,

$$\begin{aligned}\hat{q} \times \overline{d[0]\dots d[n-1]} &= \hat{q} \times \overline{d[0]\dots d[n-3]} + \beta^{n-2}\overline{d[n-2]d[n-1]}\hat{q} \\ &< \beta^{n-1} + \beta^{n-2}\overline{d[n-2]d[n-1]}\hat{q}.\end{aligned}$$

Therefore we must have $\beta^{n-2}\overline{r[0]r[1]} < \beta^{n-1}$, which implies $r[1] = 0$. □

This means that the borrow at line 18 is only non-zero when the high limb of the remainder returned by the 3-by-2 division at line 12 is zero, which is intuitively rare (if the outcomes were evenly distributed, the probability would be $1/\beta$). This lemma does not have an equivalent in the Why3 proof, as it is not needed to prove functional correctness.

Not only is the initial guess for the quotient very likely correct, but when it is not, it is only too large by 1 and we can correct it with a single incrementation. Compare this to Algorithm D, where two separate blocks were dedicated to adjusting the quotient, and one of which could be executed twice. The following lemma justifies that the candidate quotient is at worst too large by 1, which justifies that the adjustment step at line 18 is needed at most once.

LEMMA 5. *Let $\hat{q}$ and $\overline{r[0]r[1]}$ the quotient and remainder of the division of $a[n+i-2]a[n+i-1]x$ by $d[n-2]d[n-1]$.*
Then $(\hat{q}-1) \times \overline{d[0]\dots d[n-1]} \leq a[i]\dots a[n+i-1]x$.

PROOF. We have $\hat{q} \times \overline{d[n-2]d[n-1]} \leq \overline{a[n+i-2]a[n+i-1]x}$, so:

$$\begin{aligned}(\hat{q}-1)\overline{d[0]\dots d[n-1]} &\leq (\hat{q}-1)\overline{d[0]\dots d[n-3]} + \beta^{n-2}(\hat{q}-1)\overline{d[n-2]d[n-1]} \\ &< \beta^{n-1} + \beta^{n-2}\overline{a[n+i-2]a[n+i-1]x} - \beta^{n-2}\overline{d[n-2]d[n-1]} \\ &< \beta^{n-2}\overline{a[n+i-2]a[n+i-1]x} \quad \text{(we have } d[n-1] \geq \beta/2) \\ &< \overline{a[i]\dots a[n+i-1]x}. \quad \square\end{aligned}$$

The remainder of the 3-by-2 division is also used: instead of a simple long subtraction over a length n like in Algorithm D, we perform a long subtraction over a length $n-2$ only and inline the last two steps. These last two steps consist simply in propagating the borrow from the previous subtraction, as the result of the 3 most significant limbs of subtraction are known to be $\overline{lx0}$ in the absence of borrow (the postcondition of the division is exactly that $\overline{a[n+i-2]a[n+i-1]x} = \hat{q} \times \overline{d[n-2]d[n-1]} + \overline{lx}$). We then propagate the borrow on $\overline{lx0}$. Hence, lines 13 to 17 are equivalent to computing the subtraction

$$\overline{a[i]\dots a[n+i-1]x} - \hat{q} \times \overline{d[0]\dots d[n-1]}$$

returning b_2 as borrow and writing the result in $\overline{a[i]\dots a[n+i-2]x}$ (one limb fewer).

If $b_2 = 0$, the first invariant is maintained. Otherwise, there is an adjustment to make (line 18): if the subtraction overflows, our candidate quotient $\hat{q}$ was too large, we subtract 1 from it and add back $\texttt{value}(d, n)$ to the remainder. This addition is done at lines 20-21. The last limb is added separately because we save a few operations by ignoring the carry out (we know it is equal to 1).

It is efficient to do the subtraction first and potentially backtrack on it later because it happens very rarely (Lemma 4), and evaluating the subtraction makes it easy to check if the candidate quotient was correct with a simple integer comparison.

The specification of the 3-by-2 division primitive ensures that $\overline{lx} < \overline{d[n-2]d[n-1]}$, hence the second and third invariants are maintained if there is no adjustment. If there is an adjustment, the addition at line 20 may overflow, but the value of the top two limbs of the remainder will still be $\overline{lx} + 1 \leq \overline{d[n-2]d[n-1]}$. This shows that the third invariant still holds in this case. For the second invariant, we use the fact that $\texttt{value}(a+i, n-1) + \beta^{n-1}x - \beta^n b_2$ is negative if $b_2 \neq 0$, so adding back $\texttt{value}(d, n)$ maintains the second invariant.

There is an extra contingency in the main loop: when the two most significant limbs of the denominator and the current numerator are identical, then we can skip the 3-by-2 division and adjustment step. Indeed, the candidate quotient output by the division would necessarily be β, but the third invariant ensures that the long subtraction that would follow would have a non-zero borrow, and that the adjustment step would knock the candidate quotient down to $\beta - 1$. Therefore, we immediately write in $\beta - 1$ as quotient limb. With this case out of the way, we may write the 3-by-2 division in such a way that its quotient output is always a single limb, which would otherwise not be true in general.

4.2 3-by-2 division

Let us now take a closer look at the 3-by-2 division subroutine used by our division algorithm. It was introduced by Möller and Granlund [22] and is used in GMP 6.1.2.

```
predicate reciprocal_3by2 (v dh dl:limb) =
  v = div (radix*radix*radix -1) (dl + radix * dh) - radix

let div3by2_inv (uh um ul dh dl v: limb) : (limb,limb,limb)
  requires { dh ≥ div radix 2 }
  requires { reciprocal_3by2 v dh dl }
  requires { um + radix * uh < dl + radix * dh }
  returns { q, rl, rh → q * (dl + radix * dh) + rl + radix * rh
                      = ul + radix * (um + radix * uh) }
  returns { _q, rl, rh → 0 ≤ rl + radix * rh < dl + radix * dh }
```

The algorithm takes a precomputed pseudo-inverse v of the denominator $\overline{d_l d_h}$ as an extra parameter. More precisely,

$$v = \left\lfloor \frac{\beta^3 - 1}{d_l + \beta d_h} \right\rfloor - \beta.$$

The reason it is precomputed and passed as a parameter rather than computed on the fly is that the caller function (Alg. 7) always uses the same denominator over a long division, so it is much more efficient to compute the pseudo-inverse only once.

The algorithm consists essentially in multiplying $\overline{d_l d_h}$ by its pseudo-inverse v and then performing some simple adjustments. Remarkably, this means that no division primitive is used. As division primitives tend to be much more expensive than additions or multiplications, this makes the algorithm a very efficient way to perform a 3-by-2 division.

The "trick" is that the computation of the pseudo-inverse itself does use a division primitive. In fact, computing the pseudo-inverse is about as complex as the 3-by-2 division proper because of this division. However, over an m-by-n long division, $m - n$ short divisions are performed, all with the same precomputed pseudo-inverse, which amortizes that cost.

While the algorithm itself is short, its proof is non-trivial. The hardest part was directly taken from Möller and Granlund's on-paper proof [22, Theorem 3] and adapted for Why3. The algorithm that computes the pseudo-inverse with a single 2-by-1 division primitive was also taken from their paper.

4.3 Smaller cases: $n = 1$ and $n = 2$

The general case algorithm only handles the case where the denominator has length 3 or more. Different algorithms are used for smaller denominators. When the divisor is exactly one limb long, the schoolbook algorithm is used (Alg. 8).

Algorithm 8 Schoolbook division by a 1-limb number

Require: $m \geq 1, \texttt{valid}(q, m), \texttt{valid}(a, m), d \geq \beta/2$
Ensure: $\texttt{value}(a, m) = \texttt{result} + d \times \texttt{value}(q, m)$
Ensure: $\texttt{result} < d$

function DIVREM_1(q, a, m, d)
  $v \leftarrow$ INVERT_LIMB(d)
  $r \leftarrow 0$
  $i \leftarrow m - 1$
  while $i \geq 0$ **do**
    $(\hat{q}, \hat{r}) \leftarrow$ DIV2BY1_INV($r, a[i], d, v$) ▷ Divide $\overline{a[i]r}$ by d
    $q[i] \leftarrow q$
    $r \leftarrow \hat{r}$
    $i \leftarrow i - 1$
  return r

The variable r is the partial remainder, in the sense of the following loop invariants:

(1) $\texttt{value}(a + i + 1, m - i - 1) = d \times \texttt{value}(q + i + 1, m - i - 1) + r$
(2) $r < \beta$.

Let us prove that the first invariant is maintained. Assume $\texttt{value}(a + i + 1, m - i - 1) = d \times \texttt{value}(q + i + 1, m - i - 1) + r$ and $r < \beta$. Let $\hat{q}, \hat{r}$ such that $\hat{q}d + \hat{r} = a[i] + \beta r$. Then after $q[i] \leftarrow q$,

$$
\begin{aligned}
&\texttt{value}(a + (i-1) + 1, m - (i-1) - 1) \\
&= \texttt{value}(a + i, m - i) \\
&= a[i] + \beta\texttt{value}(a + i + 1, m - i - 1) && \texttt{[value\_sub\_head]} \\
&= a[i] + \beta d \times \texttt{value}(q + i + 1, m - i - 1) + \beta r \\
&= \hat{q}d + \hat{r} + \beta d \times \texttt{value}(q + i + 1, m - i - 1) \\
&= d \times \texttt{value}(q + i, m - i) + \hat{r} \\
&= d \times \texttt{value}(q + (i-1) + 1, m - (i-1) - 1) + \hat{r}.
\end{aligned}
$$

Finally, similarly to the general algorithm, we precompute a pseudo-inverse v of d for the 2-by-1 division, and subsequently use a 2-by-1 division algorithm that uses no division primitive. This time, we have $v = \left\lfloor \frac{\beta^2 - 1}{d} \right\rfloor - \beta$.

```
predicate reciprocal (v d:limb) =
  v = (div (radix*radix - 1) (d)) - radix

let div2by1_inv (uh ul d v:limb) : (limb,limb)
  requires { d ≥ div radix 2 }
  requires { uh < d }
  requires { reciprocal v d }
  returns { q, r →  q * d + r = ul + radix * uh }
  returns { q, r → 0 ≤ r < d }
```

When $n = 2$, a very similar algorithm is used (Alg. 9). The only difference is that 3-by-2 division is used at each loop iteration. Similarly to the $n = 1$ case, $\overline{lh}$ is the current

Algorithm 9 Schoolbook division by a 2-limb number

Require: $m \geq 2, \texttt{valid}(q, m-2), \texttt{valid}(a, m), \texttt{valid}(d, 2), d[1] \geq \beta/2$
Require: $\texttt{value}(a+m-2, 2) < \texttt{value}(d, 2)$ ▷ Otherwise an extra quotient limb is needed
Ensure: $\texttt{value}(\texttt{old } a, m) = \texttt{value}(d, 2) \times \texttt{value}(q, m-2) + \texttt{value}(a, 2)$
Ensure: $\texttt{value}(a, 2) < \texttt{value}(d, 2)$

```
1:  function DIVREM_2(q, a, m, d)
2:      v ← RECIPROCAL_WORD_3BY2(d[1], d[0])
3:      h ← a[m − 1]
4:      l ← a[m − 2]
5:      i ← m − 2
6:      while i > 0 do
7:          (q̂, l, h) ← DIV3BY2_INV(h, l, a[i − 1], d[1], d[0], v)
8:          i ← i − 1
9:          q[i] ← q̂
10:     a[1] ← h
11:     a[0] ← l
```

remainder, and the loop invariants are as follows:

$$\texttt{value}(a, m) = \texttt{value}(a, i) + \beta^i(\texttt{value}(q+i, m-i-2) \times \texttt{value}(d, 2) + l + \beta h)$$
$$l + \beta h < \texttt{value}(d, 2).$$

Let us now discuss the differences between these two special cases on one hand, and the general case algorithm in the other hand. The adjustment step that is found in Algorithm D and other general long division algorithm is not needed in any of these two special cases. Indeed, the source of error in Algorithm D (and in GMP's algorithm) that makes the adjustment step necessary is that the candidate quotient is computed using an incomplete denominator. When the denominator has length 1 or 2, the division that occurs at each loop iteration uses the full denominator. Even though the numerator still has arbitrary length, this does not cause errors on the candidate quotient.

Another difference is that no subtraction is needed to compute the partial remainders in the two short cases. The corresponding operation in Alg. 8 is simply the $r \leftarrow \hat{r}$ assignment at line 8. Indeed, we have $d\hat{q}+\hat{r} = \overline{a[i]r}$ from the 2-by-1 division, and the long subtraction that we would perform with Algorithm D would be exactly $\overline{a[i]r}-d\hat{q} = \hat{r}$, so decrementing i and assigning r to $\hat{r}$ does the same thing and saves a subtraction. Again, this is due to the fact that we use the whole denominator at each loop iteration.

5. TOOM-COOK MULTIPLICATION

The schoolbook multiplication algorithm from Section 3.3 has quadratic complexity and is only optimal for numbers shorter than about 30 limbs, or 2000 bits. For larger numbers (between 2000 and about 100,000 bits), GMP uses a family of recursive multiplication algorithms initially introduced by Toom [27] and Cook [8]. These algorithms for integer and polynomial multiplication can be viewed as solving a multipoint evaluation and polynomial interpolation problem.

The general principle of Toom-Cook algorithms is to choose a base B, typically a power of 2^{64}, and to view the digits of the factors in base B as coefficients of polynomials a

and b. We then evaluate those polynomials at well-chosen points v_i, compute the products $a(v_i)b(v_i)$ by calling the algorithm recursively, and interpolate to obtain the coefficients of the product polynomial c. The product is then obtained by evaluating $c(B)$.

We have verified two Toom-Cook algorithms: Toom-2 (Sec. 5.1), which is similar to Karatsuba multiplication [18], and its unbalanced variant Toom-2.5 (Sec. 5.2), introduced by Bodrato and Zanoni [5].

Toom-2 (also called `toom22` in the code) splits each of the operands into two parts roughly equal in length, and Toom-2.5 (or `toom32`) splits the largest operand into three parts and the smallest into two. Toom-2 is called on operands of roughly equal length and Toom-2.5 is called when one of the operands is about 1.5 times as long as the other. This way, after splitting, we are left with parts that have roughly equal length. A general case algorithm, which we describe in Sec. 5.3, reduces all cases to applications of the two former ones.

5.1 Toom-2

Algorithm 10 Toom-2 multiplication

Require: $2 \leq n \leq m < 30 \times 2^k$
Require: $\texttt{valid}(r, m+n), \texttt{valid}(a, m), \texttt{valid}(b, n), \texttt{valid}(s, 2(m+k))$
Require: $4 \cdot m < 5 \cdot n$
Ensure: $\texttt{value}(r, m+n) = \texttt{value}(a, m) \times \texttt{value}(b, n)$

1: **function** TOOM22_MUL(r, a, b, s, m, n, k)
2: $\mu \leftarrow m \gg 1$
3: $\lambda \leftarrow m - \mu$
4: $\nu \leftarrow n - \lambda$
5: $(a_0, a_1) \leftarrow (a, a + \lambda)$
6: $(b_0, b_1) \leftarrow (b, b + \lambda)$
7: Compute $|a(-1)|$ in r, $|b(-1)|$ in $r + \lambda$, sign in ϵ (see Alg. 12)
8: $(c_0, c_\infty) \leftarrow (r, r + 2\lambda)$
9: $s' \leftarrow s + 2\lambda$
10: TOOM22_MUL_REC($s, r, r + \lambda, s', \lambda, \lambda, k - 1$) ▷ Compute $|c(-1)|$ recursively.
11: TOOM22_MUL_REC($c_\infty, a_1, b_1, s', \mu, \nu, k - 1$) ▷ Compute $c(+\infty)$ recursively.
12: TOOM22_MUL_REC($c_0, a_0, b_0, s', \lambda, \lambda, k - 1$) ▷ Compute $c(0)$ recursively.
13: $v \leftarrow$ ADD_N($c_\infty, c_0 + \lambda, c_\infty, \lambda$) ▷ $H_0 + L_\infty$
14: $v_2 \leftarrow v +$ ADD_N($c_0 + \lambda, c_\infty, c_0, \lambda$) ▷ $L_0 + H_0 + L_\infty$
15: $v \leftarrow v +$ ADD($c_\infty, c_\infty, \lambda, c_\infty + \lambda, \mu + \nu - \lambda$) ▷ $H_0 + L_\infty + H_\infty$
16: **if** $\epsilon = 1$ **then**
17: $v \leftarrow v +$ ADD($r + \lambda, r + \lambda, c_{-1}, 2\lambda$)
18: **else** $v \leftarrow v -$ SUB($r + \lambda, r + \lambda, c_{-1}, 2\lambda$) $\mod \beta$ ▷ $v \in \{0, 1, 2, \beta - 1\}$.
19: INCR(c_∞, v_2)
20: **if** $v \leq 2$ **then** ▷ Implies $0 \leq v$.
21: INCR($r + 3\lambda, v$)
22: **else** DECR($r + 3\lambda, 1$) ▷ $v = \beta - 1$ instead of -1, due to integer representation.

Algorithm 11 Recursive call in Toom-2

Require: $\texttt{valid}(r, m+n), \texttt{valid}(a, m), \texttt{valid}(b, n), \texttt{valid}(s, 2(m+k))$
Require: $4m < 5n$
Ensure: $\texttt{value}(r, m+n) = \texttt{value}(a, m) \times \texttt{value}(b, n)$
  function TOOM22_MUL_REC(r, a, b, s, m, n, k)
    if $b < 30$ **then** ▷ Operands are small, use the schoolbook algorithm.
      MUL_BASECASE(r, a, b, m, n)
    else
      if $4 \cdot m < 5 \cdot n$ **then**
        TOOM22_MUL(r, a, b, s, m, n, k)
      else ▷ Operands are unbalanced, use Toom-2.5.
        TOOM32_MUL(r, a, b, s, m, n, k)

Algorithm 12 Computation of $|a(-1)|$ in r and $|b(-1)|$ in $r + \lambda$

Ensure: $\texttt{value}(r, \lambda) = |a(-1)|$
Ensure: $\texttt{value}(r + \lambda, \lambda) = |b(-1)|$
Ensure: $\epsilon \cdot \texttt{value}(r, \lambda) \cdot \texttt{value}(r + \lambda, \lambda) = a(-1)b(-1)$
  $\epsilon \leftarrow 1$ ▷ Will hold the sign of $a(-1)b(-1)$.
  if $\lambda = \mu$ **then** ▷ Compute $a(-1)$.
    if COMPARE$(a_0, a_1, \lambda) < 0$ **then** ▷ $A_1 > A_0$
      SUB(r, a_1, a_0, λ)
      $\epsilon \leftarrow -1$
    else SUB(r, a_0, a_1, λ)
  else ▷ $\lambda = \mu + 1$
    if $a_0[\mu] = 0 \wedge$ COMPARE$(a_0, a_1, \mu) < 0$ **then** ▷ $A_1 > A_0$
      SUB(r, a_1, a_0, μ)
      $r[\mu] \leftarrow 0$
      $\epsilon \leftarrow -1$
    else
      $t \leftarrow$ SUB(r, a_0, a_1, μ)
      $r[\mu] \leftarrow a_0[\mu] - t$ ▷ No carry, as we know $a_0 \geq a_1$.
  if $\lambda = \nu$ **then** ▷ Compute $b(-1)$.
    if COMPARE$(b_0, b_1, \lambda) < 0$ **then** ▷ $B_1 > B_0$
      SUB$(r + \lambda, b_1, b_0, \lambda)$
      $\epsilon \leftarrow -\epsilon$ ▷ Change the sign of $a(-1)b(-1)$.
    else SUB$(r + \lambda, b_0, b_1, \lambda)$
  else
    if IS_ZERO$(b_0 + \nu, \lambda - \nu) \wedge$ COMPARE$(b_0, b_1, \nu) < 0$ **then** ▷ $B_1 > B_0$
      SUB$(r + \lambda, b_1, b_0, \nu)$ ▷ b_0 also has length at most ν.
      ZERO$(r + \lambda + \nu, \lambda - \nu)$ ▷ We still have to initialize the rest of $(r + \lambda, \lambda)$.
      $\epsilon \leftarrow -\epsilon$
    else SUB$(r + \lambda, b_0, \lambda, b_1, \nu)$

The parameters of the `toom22_mul` function (Alg. 10) are as follows: r is the destination buffer, a and b are the source operands, m and n are their lengths in limbs, and s is an extra buffer to store temporary results.

The amount of space needed for s is approximately $2(m + \log_2(m))$ limbs. Rather than explicitly talking about logarithms in the specification, we use an extra parameter $k \geq \log_2(m)$. The variable k is *ghost* [12], which means that it is never used in the computations, but only in the proof. The first precondition ensures that k is a suitable bound. In practice, the caller of Toom-2 can give $k = 64$ (and allocate $2m + 128$ limbs as scratch space), as integer lengths are machine integers, so smaller than 2^{64}.

The last precondition makes sure that the operands are sufficiently close in size. It could be a bit looser without breaking the algorithm, but the unbalanced version Toom-2.5 is more efficient when the operand sizes are too unbalanced.

The algorithm is organized in four steps. First we split the operands into two parts of roughly equal length (Sec. 5.1.1). Then we evaluate the product polynomial c at three points (Sec. 5.1.2). We then recompose the coefficients of c through interpolation (Sec. 5.1.3). Finally, we propagate the remaining carries (Sec. 5.1.4).

5.1.1 *Splitting (Alg. 10, lines 2-6).* We pose $\mu = \lfloor \frac{m}{2} \rfloor$, $\lambda = m - \mu$, $\nu = n - \lambda$. The preconditions ensure the following:

$$\begin{aligned} 0 < \nu \leq \mu \leq \lambda < m \\ \lambda - 1 \leq \mu \leq \lambda \\ \lambda < \mu + \nu \end{aligned}$$

We can split a into two subwords a_0 and a_1 such that $\texttt{value}(a, m) = \texttt{value}(a_0, \lambda) + \beta^\lambda \times \texttt{value}(a_1, \mu)$. Similarly we have b_0 and b_1 such that $\texttt{value}(b, n) = \texttt{value}(b_0, \lambda) + \beta^\lambda \times \texttt{value}(b_1, \nu)$.

We denote $A_0 := \texttt{value}(a_0, \lambda)$ and so on. Likewise, we define the polynomials $a(X) := A_0 + A_1 X$ and $b(X) := B_0 + B_1 X$. The goal is to compute $c(\beta^\lambda)$, where $c(X) = a(X)b(X)$ is a degree-2 polynomial. We pose $c(X) = C_0 + C_1 X + C_2 X^2$.

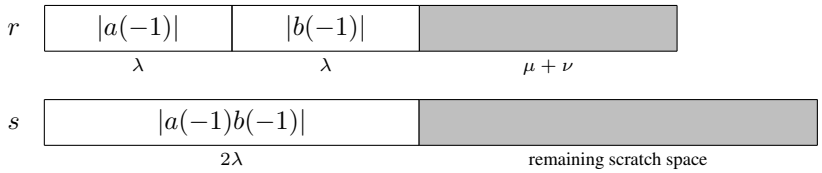
5.1.2 *Evaluation (Alg. 10, lines 7-12).* The first step is to obtain three values of $c(X)$ for interpolation. GMP chooses to evaluate c at 0, -1 and $+\infty$ (where $c(+\infty)$ is defined as C_2). We first evaluate $a(-1) = A_0 - A_1$ and $b(-1) = B_0 - B_1$ (Alg. 12). To avoid carry propagations, we first check which of A_0 and A_1 is larger to compute $|a(-1)|$ and store its sign separately in a variable ϵ. If $\mu = \lambda - 1$, we can optimise slightly by performing a subtraction of length μ instead of λ.

Similarly, we compute $|b(-1)|$ and update ϵ to contain the sign of $a(-1)b(-1)$. We store $|a(-1)|$ in the first λ limbs of r (we denote this subarray $r(0, \lambda)$) and $|b(-1)|$ in the next λ limbs (which we denote $r(\lambda, 2\lambda)$). We then call Toom-2 recursively to compute $|a(-1)b(-1)|$ and store the result in $s(0, 2\lambda)$ (Alg. 10, line 10). We use $s(2\lambda, \ldots)$ as scratch space (there is enough space because k decreased by one).

The constant 30 in the recursive call function TOOM22_MUL_REC (Alg. 11) is the minimum integer length for which our library calls Toom-Cook multiplication algorithms. For

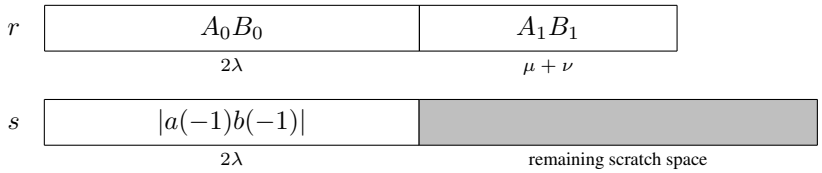
smaller numbers, the schoolbook multiplication is called instead. The value 30 was picked experimentally, and the optimum can vary from machine to machine.

After the recursive call, the memory layout is as follows. Throughout this section, the memory layout will be illustrated with such diagrams (in addition to the formulas) to help visualise the current state of the computation.



The next step is to compute $c(+\infty)$, that is, A_1B_1. This is done with a simple recursive call to Toom-2 (Alg. 10, line 11), using $s(2\lambda, \ldots)$ as scratch space again. The result has size $\mu + \nu$, which fits in $r(2\lambda, 2\lambda + \mu + \nu)$.

Finally, we compute $c(0)$, that is, A_0B_0 (line 12). We use the second half of s as scratch space again, and store the result in $r(0, 2\lambda)$, writing over $|a(-1)|$ and $|b(-1)|$ (but their product still is in the first half of s).

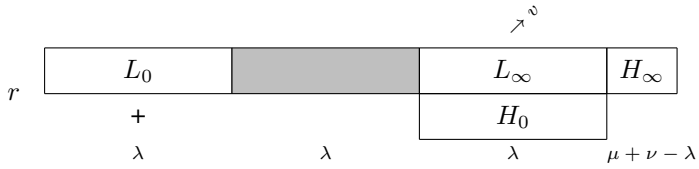


We further decompose $c(0)$, $c(-1)$ and $c(+\infty)$ in halves of size λ or less:

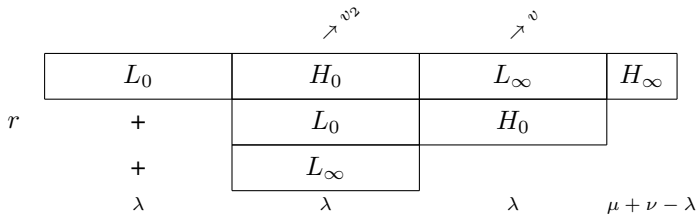
$$A_0B_0 = c(0) = L_0 + \beta^\lambda \times H_0$$
$$A_1B_1 = c(+\infty) = L_\infty + \beta^\lambda \times H_\infty.$$

At the end of the evaluation step, we have the following memory layout:

5.1.3 *Recomposition (Alg. 10, lines 13-15).* We first add H_0 to L_∞ in place, storing the carry in a variable v (line 13). The outgoing up-right arrows in the diagrams that follow represent the carry out.

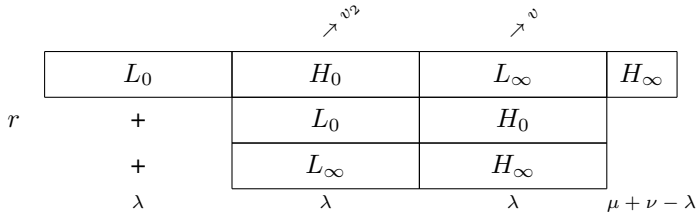


We then add the result to L_0, writing over the original location of H_0 (line 14). We store the sum of v and the new carry in v_2. At that point, $\texttt{value}(r+\lambda,\lambda)+\beta^\lambda v_2 = L_0+H_0+L_\infty$.



Finally, we add H_∞ to $r(2\lambda,3\lambda)$ in place, incrementing v if needed (line 15).
At that point, we have:

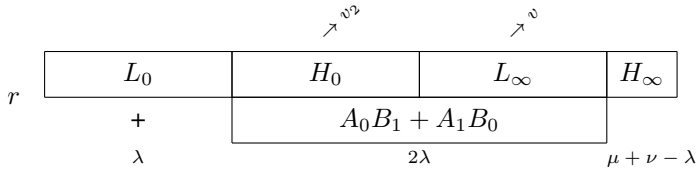
$$\begin{aligned}\texttt{value}(r+\lambda,2\lambda)+\beta^\lambda v_2+\beta^{2\lambda}v &= (H_0+L_0+L_\infty)+\beta^\lambda(H_0+L_\infty+H_\infty)\\ &= A_0B_0+A_1B_1+H_0+\beta^\lambda L_\infty. \end{aligned} \tag{1}$$



Finally, we subtract $c(-1)$ from $r(\lambda,3\lambda)$ by adding or subtracting $|a(-1)||b(-1)|$, depending on the stored sign (lines 16-18).

At that point, we have $-1 \le v \le 3$, and:

$$\begin{aligned}\texttt{value}(r+\lambda,2\lambda)+\beta^\lambda v_2+\beta^{2\lambda}v &= A_0B_0+A_1B_1+H_0+\beta^\lambda L_\infty-(A_0-A_1)(B_0-B_1)\\ &= A_0B_1+A_1B_0+H_0+\beta^\lambda L_\infty.\end{aligned}$$



Therefore,

$$\begin{aligned}&\texttt{value}(r,m+n)+\beta^{2\lambda}v_2+\beta^{3\lambda}v\\ &= L_0+\beta^\lambda(\texttt{value}(r+\lambda,2\lambda)+\beta^\lambda v_2+\beta^{2\lambda}v)+\beta^{3\lambda}H_\infty\\ &= A_0B_0+\beta^\lambda(A_0B_1+A_1B_0)+\beta^{2\lambda}A_1B_1\\ &= (A_0+\beta^\lambda A_1)(B_0+\beta^\lambda B_1).\end{aligned}$$

The only thing left to do is propagating the carries.

5.1.4 *Carry propagation (Alg. 10, lines 19-22).* We first propagate v_2 at line 19, then v (lines 20-22). There is an `if` statement because the case $v_2 = -1$ (represented as the unsigned integer $\beta - 1$) requires special treatment, as $\text{INCR}(\cdot, \beta - 1)$ is not the same thing as $\text{DECR}(\cdot, 1)$. Another thing to look out for is that the functions INCR and DECR that propagate the carries never check whether they reach the bounds of the array that holds the number to be incremented. Rather, they perform an addition (or subtraction) and propagate the carry until there is none. Their preconditions include the fact that this computation should not overflow. Calling these functions incorrectly could result in buffer overflows. This caused various bugs in past versions of GMP where this precondition was mistakenly violated in rare cases. Notably, this is a situation where the memory safety of the program (absence of buffer overflow) directly depends on its functional correctness (the number that is being incremented fits between certain bounds).

The most difficult part of the whole Toom-2 proof is ensuring that the propagation of the carries v and v_2 does not overflow. This is easy in the case where $v \geq 0$, as the total product ab (obtained after propagating both carries) is certain to fit in $m + n$ and the intermediate value obtained after propagating one of the carries is certain to be between 0 and ab.

The only nontrivial case is $v = -1, v_2 \neq 0$. In this case, it is not obvious that the first propagation, that of v_2, does not overflow out of r, as we have:

$$\texttt{value}(r, m + n) = (A_0 + \beta^\lambda A_1)(B_0 + \beta^\lambda B_1) - \beta^{2\lambda} v_2 + \beta^{3\lambda},$$

and we might have $(A_0 + \beta^\lambda A_1)(B_0 + \beta^\lambda B_1) + \beta^{3\lambda} \geq \beta^{m+n}$.

It is sufficient to show that $H_\infty < \beta^{\mu+\nu-\lambda} - 1$, that is, that the binary representation of H_∞ is not all ones. Indeed, the carry is absorbed somewhere in H_∞ if it is the case.

If $v = -1$, the first addition (of H_0 and L_∞) cannot overflow, and we must have $v_2 \leq 1$. The only case to consider is therefore $v = -1$, $v_2 = 1$.

If $v = -1$, the substraction of c_{-1} from $r + \lambda$ necessarily underflows, that is,

$$A_0 B_0 + A_1 B_1 + H_0 + \beta^\lambda L_\infty - \beta^\lambda < (A_0 - A_1)(B_0 - B_1)$$

(Equation (1) with $v_2 = 1$, $v = 0$).

Noticing that $0 \leq H_0$ and $0 \leq A_0 B_1 + A_1 B_0 = A_0 B_0 + A_1 B_1 - (A_0 - A_1)(B_0 - B_1)$, we are left with $\beta^\lambda L_\infty - \beta^\lambda < 0$, which implies $L_\infty = 0$.

Let us pose x, y the 2-adic valuations of A_1 and B_1, and a', b' odd integers such that $A_1 = 2^x a'$, $B_1 = 2^y b'$. We have $2^{x+y} a' b' = A_1 B_1 = \beta^\lambda H_\infty = 2^{64\lambda} H_\infty$. As a' and b' are odd we must have $x + y \geq 64\lambda$. If H_∞ is even we are done, so we can assume $x + y = 64\lambda$ and $a'b' = H_\infty$ without loss of generality.

Notice now that $A_1 < 2^{64\mu}$ as it fits in a zone of length μ. We therefore have $x < 64\mu$ and $a' \leq 2^{64\mu - x} - 1$, similarly $y < 64\nu$ and $b' \leq 2^{64\nu - y} - 1$.

Therefore,

$$\begin{aligned} H_\infty = a'b' &\leq (2^{64\mu - x} - 1)(2^{64\nu - y} - 1) \\ &= \beta^{\mu+\nu-\lambda} - 2^{64\nu - y} - 2^{64\mu - x} + 1 \\ &\leq \beta^{\mu+\nu-\lambda} - 3, \end{aligned}$$

and we can conclude that the propagation of v_2 does not overflow.

We did not expect this fact to require such a complex proof, especially because there were no comments in the GMP source code to indicate that non-trivial reasoning was

needed. We discussed this with the developers, and they ended up changing the algorithm to make it more clearly correct at no performance cost.[4]

5.2 Toom-2.5

Toom-2.5, also known as `toom32`, has the same signature as Toom-2. The algorithm is similar (Alg. 13), but splits the larger operand into three parts rather than two. It is optimally used when the first operand is longer than the second by half.

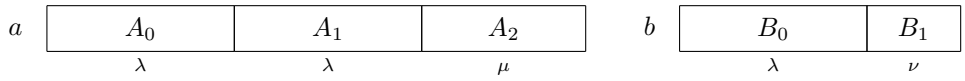
The algorithm is organized into phases that are similar to those of Toom-2. First we split the larger operand into three parts and the smaller one into two parts (Sec. 5.2.1). Then we evaluate c at four points (Sec. 5.2.2 and 5.2.4), and finally we recompose the coefficients of c (Sec. 5.2.3 and 5.2.5).

5.2.1 *Splitting (Alg. 13, lines 2-9).* We pose either $\lambda = 1 + \lfloor \frac{m-1}{3} \rfloor$ or $\lambda = 1 + \lfloor \frac{n-1}{2} \rfloor$, whichever is longest. We also pose $\mu = m - 2\lambda$ and $\nu = n - \lambda$. The preconditions ensure that $0 < \nu \leq \lambda$, $0 < \mu \leq \lambda$, and $\lambda \leq \mu + \nu$.

Similarly to Toom-2, we split a into three subwords a_0, a_1 and a_2 of lengths λ, λ and μ respectively. We also split b into two subwords b_0 and b_1 of lengths λ and ν. We denote $A_0 := \texttt{value}(a_0, \lambda)$ and so on. We define the polynomials $a(X) := A_0 + A_1 X + A_2 X^2$ and $b(X) := B_0 + B_1 X$. The goal is to compute $c(\beta^\lambda)$, where $c(X) = a(X)b(X)$ is a degree-3 polynomial. We pose $c(X) = C_0 + C_1 X + C_2 X^2 + C_3 X^3$.

$$
\begin{aligned}
C_0 &= A_0 B_0 \\
C_1 &= A_1 B_0 + A_0 B_1 \\
C_2 &= A_2 B_0 + A_1 B_1 \\
C_3 &= A_2 B_1
\end{aligned}
$$

The lengths of A_i and B_i coefficients imply that C_0 has length 2λ, C_1 and C_2 have length $2\lambda + 1$ and C_3 has length $\mu + \nu$.



5.2.2 *Evaluation in* 1 *and* -1 *(Alg. 13, lines 10-25, and full Alg. 14).* As $c(X)$ has degree 3, this time we need to obtain four values for interpolation. GMP chooses to evaluate c at $0, -1, 1$ and $+\infty$. We remark that $a(-1) = A_0 - A_1 + A_2$ and $a(1) = A_0 + A_1 + A_2$. Therefore, some evaluation steps can be saved by computing $A_0 + A_2$ only once and using this result for both evaluations (Alg. 14). Just like in Toom-2, we actually compute $|a(-1)|$ and $|b(-1)|$ and store the sign of the product separately in a variable ϵ. The carries of the evaluation of $a(1)$, $b(1)$ and $|a(-1)|$ are stored in $a^\sharp$, $b^\sharp$ and $a^\flat$ respectively. The evaluation of $|b(-1)| = |B_0 - B_1|$ cannot overflow (and this is why we bother computing absolute values and storing the sign separately).

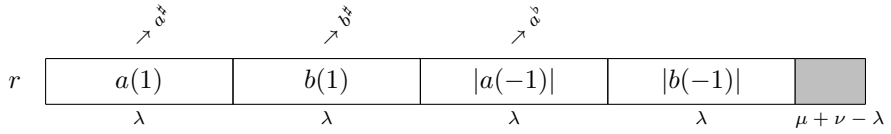
[4] `https://gmplib.org/repo/gmp/rev/02a2ec6e1bce`

Algorithm 13 Toom-2.5 multiplication

Require: $30 \leq n \leq m < 30 \times 2^k$
Require: $\texttt{valid}(r, m+n), \texttt{valid}(a, m), \texttt{valid}(b, n), \texttt{valid}(s, 2(m+k))$
Require: $n + 2 \leq m \wedge m + 6 \leq 3 \cdot n$
Ensure: $\texttt{value}(r, m+n) = \texttt{value}(a, m) \times \texttt{value}(b, n)$

1: **function** TOOM32_MUL(r, a, b, s, m, n, k)
2: **if** $2m \geq 3n$ **then**
3: $\lambda \leftarrow 1 + \lfloor \frac{m-1}{3} \rfloor$
4: **else**
5: $\lambda \leftarrow 1 + \lfloor \frac{n-1}{2} \rfloor$
6: $\mu \leftarrow m - 2\lambda$
7: $\nu \leftarrow n - \lambda$
8: $(a_0, a_1, a_2) \leftarrow (a, a + \lambda, a + 2\lambda)$
9: $(b_0, b_1) \leftarrow (b, b + \lambda)$
Compute $a(1)$, $b(1)$, $|a(-1)|$, $|b(-1)|$ in r, sign in ϵ (see Alg. 14).
Carries of $a(1)$ and $b(1)$ are in $a^\sharp$ and $b^\sharp$, carry of $|a(-1)|$ in $a^\flat$.
10: $s' \leftarrow s + 2\lambda + 1$
11: TOOM22_MUL_REC($s, r, r + \lambda, s', \lambda, \lambda, k - 1$) ▷ Compute $c(1)$ recursively.
12: **if** $a^\sharp = 1$ **then** ▷ Propagate the carry for $c(1)$.
13: $v \leftarrow b^\sharp + \text{ADD\_N}(s + \lambda, s + \lambda, r + \lambda, \lambda)$
14: **else**
15: **if** $a^\sharp = 2$ **then**
16: $v \leftarrow 2b^\sharp + \text{ADDMUL\_1}(s + \lambda, r + \lambda, \lambda, 2)$
17: **else** ▷ $a^\sharp = 0$
18: $v \leftarrow 0$
19: **if** $b^\sharp$ **then**
20: $v \leftarrow v + \text{ADD\_N}(s + \lambda, s + \lambda, r, \lambda)$
21: $s[2\lambda] \leftarrow v$
22: TOOM22_MUL_REC($r, r + 2\lambda, r + 3\lambda, s', \lambda, \lambda, k - 1$) ▷ Compute $|c(-1)|$ recursively.
23: **if** $a^\flat$ **then** ▷ Propagate the carry for $|c(-1)|$.
24: $a^\flat \leftarrow \text{ADD\_N}(r + \lambda, r + \lambda, r + 3\lambda, \lambda)$
25: $r[2\lambda] \leftarrow a^\flat$
26: **if** $\epsilon = -1$ **then**
27: SUB_N($s, s, r, 2\lambda + 1$)
28: **else**
29: ADD_N($s, s, r, 2\lambda + 1$)
30: RSHIFT($s, s, 2\lambda + 1, 1$) ▷ $s \leftarrow \frac{c(1)+c(-1)}{2} = C_0 + C_2$
31: $v \leftarrow \text{ADD\_N}(r + 2\lambda, s, s + \lambda, \lambda)$ ▷ Add L and M in D_1.
32: INCR($s + \lambda, \lambda + 1, v + s[2\lambda]$) ▷ Propagate v and h to D_2.

33: **if** $\epsilon = -1$ **then**
34: $v \leftarrow \text{ADD\_N}(s, s, r, \lambda)$
35: $v' \leftarrow r[2\lambda] + \text{ADD\_NC}(r + 2\lambda, r + 2\lambda, r + \lambda, \lambda, v)$
36: $\text{INCR}(s + \lambda, \lambda + 1, v')$
37: **else**
38: $v \leftarrow \text{SUB\_N}(s, s, r, \lambda)$
39: $v' \leftarrow r[2\lambda] + \text{SUB\_NC}(r + 2\lambda, r + 2\lambda, r + \lambda, \lambda, v)$
40: $\text{DECR}(s + \lambda, \lambda + 1, v')$
41: $\text{TOOM22\_MUL\_REC}(r, a_0, b_0, s', \lambda, \lambda, k - 1)$ ▷ Compute $c(0)$ recursively.
42: **if** $\mu > \nu$ **then** ▷ Compute $c(+\infty)$ recursively.
43: $\text{MUL}(r + 3\lambda, a_2, \mu, b_1, \nu)$
44: **else**
45: $\text{MUL}(r + 3\lambda), b_1, \nu, a_2, \mu)$
46: $v \leftarrow \text{SUB\_N}(r + \lambda, r + \lambda, r + 3\lambda, \lambda)$
47: $v' \leftarrow s[2\lambda] + v$
48: $v \leftarrow \text{SUB\_NC}(r + 2\lambda, r + 2\lambda, r, \lambda, v)$
49: $v' \leftarrow v' - \text{SUB\_NC}(r + 3\lambda, s + \lambda, r + \lambda, \lambda, v)$
50: $v' \leftarrow v' + \text{ADD}(r + \lambda, r + \lambda, 3\lambda, s, \lambda)$
51: **if** $\mu + \nu > \lambda$ **then** ▷ Propagate v'.
52: $v' \leftarrow v' - \text{SUB}(r + 2\lambda, r + 2\lambda, 2\lambda, r + 4\lambda, \mu + \nu - \lambda)$
53: **if** $v' < 0$ **then**
54: $\text{DECR}(r + 4\lambda, \mu + \nu - \lambda, -v')$
55: **else**
56: $\text{INCR}(r + 4\lambda, \mu + \nu - \lambda, v')$



After this step, we have $a(1) = \texttt{value}(r, \lambda) + \beta^\lambda a^\sharp$ and so on. A simple case analysis shows $0 \leq a^\sharp \leq 2$, $0 \leq b^\sharp \leq 1$, and $0 \leq a^\flat \leq 1$. The next thing to do is to compute $c(1)$.

$$c(1) = a(1)b(1) = (\texttt{value}(r, \lambda) + \beta^\lambda a^\sharp) \cdot (\texttt{value}(r + \lambda, \lambda) + \beta^\lambda b^\sharp)$$

The recursive call at line 11 of Alg. 13 computes $\texttt{value}(r, \lambda) \cdot \texttt{value}(r + \lambda, \lambda)$ in s. The `if` statement at line 12 adds $a^\sharp \texttt{value}(r + \lambda, \lambda)$, shifted by λ as this term must be multiplied by β^λ. It also accumulates the carry and $a^\sharp b^\sharp$ into the variable v. After this, we have

$$\texttt{value}(s, 2\lambda) + \beta^{2\lambda} v = (\texttt{value}(r, \lambda) + \beta^\lambda a^\sharp) \cdot \texttt{value}(r + \lambda, \lambda) + \beta^{2\lambda} a^\sharp b^\sharp$$

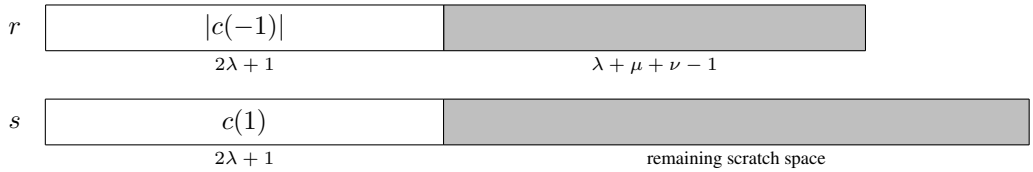
The only missing term is $\beta^\lambda b^\sharp \cdot \texttt{value}(r, \lambda)$, which is handled in the `if` statement at line 16 of Alg. 13. After this, we have $\texttt{value}(s, 2\lambda) + \beta^{2\lambda} v = c(1)$, and we can set $s[2\lambda]$ to v.

The product $|c(-1)| = |a(-1)b(-1)|$ is computed similarly at lines 22-29. As there is no carry corresponding to $|b(-1)|$, the procedure is a bit simpler. The term $\beta^\lambda a^\flat \cdot \texttt{value}(r +$

Algorithm 14 Toom-2.5: evaluation in 1 and -1

Ensure: $\texttt{value}(r, \lambda) + \beta^\lambda a^\sharp = a(1)$
Ensure: $\texttt{value}(r + \lambda, \lambda) + \beta^\lambda b^\sharp = b(1)$
Ensure: $\texttt{value}(r + 2\lambda, \lambda) + \beta^\lambda a^\flat = |a(-1)|$
Ensure: $\texttt{value}(r + 3\lambda, \lambda) = |b(-1)|$
Ensure: $\epsilon \cdot (\texttt{value}(r + 2\lambda, \lambda) + \beta^\lambda a^\flat) \cdot \texttt{value}(r + 3\lambda, \lambda) = a(-1)b(-1)$
  $a' \leftarrow \text{ADD}(r, a_0, \lambda, a_2, \mu)$
  if $a' = 0 \wedge \text{CMP}(r, a_1, \lambda) < 0$ **then** ▷ $A_0 + A_2 < A_1$
    $\text{SUB\_N}(r + 2\lambda, a_1, r, \lambda)$
    $a^\flat \leftarrow 0$
    $\epsilon \leftarrow -1$
  else ▷ $A_1 \leq A_0 + A_2$
    $v \leftarrow \text{SUB\_N}(r + 2\lambda, r, a_1, \lambda)$
    $a^\flat \leftarrow a' - v$
    $\epsilon \leftarrow 1$
  $a^\sharp \leftarrow a' + \text{ADD\_N}(r, r, a_1, \lambda)$ ▷ Finish computing $a(1)$.
  if $\lambda = \nu$ **then**
    $b^\sharp \leftarrow \text{ADD\_N}(r + \lambda, b_0, b_1, \lambda)$
    if $\text{CMP}(b_0, b_1, \lambda) < 0$ **then** ▷ $B_0 < B_1$
      $\text{SUB\_N}(r + 3\lambda, b_1, b_0, \lambda)$
      $\epsilon \leftarrow -\epsilon$
    else
      $\text{SUB\_N}(r + 3\lambda, b_0, b_1, \lambda)$
  else
    $b^\sharp \leftarrow \text{ADD}(r + \lambda, b_0, b_1, \lambda, \nu)$
    if $\text{IS\_ZERO}(b_0 + \nu, \lambda - \nu) \wedge \text{CMP}(b_0, b_1, \nu) < 0$ **then** ▷ $B_0 < B_1$
      $\text{SUB\_N}(r + 3\lambda, b_1, b_0, \nu)$ ▷ b_0 also has length at most ν.
      $\text{ZERO}(r + 3\lambda + \nu, \lambda - \nu)$ ▷ We still have to initialize the rest of $(r + 3\lambda, \lambda)$.
      $\epsilon \leftarrow -\epsilon$
    else
      $\text{SUB}(r + 3\lambda, b_0, b_1, \lambda, \nu)$

$3\lambda, \lambda)$ is added in the `if` statement at line 23. The product is stored in $r(0, 2\lambda + 1)$. This overwrites $a(1)$, $b(1)$ and part of $|a(-1)|$, but these intermediate results are no longer needed.

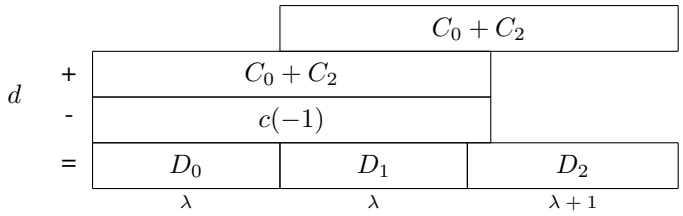


5.2.3 *Recomposition (Alg. 13, lines 26-40).* We use the intermediate variable $d := C_1 + C_3 + \beta^\lambda \cdot (C_0 + C_2)$. The sizes of the C_i imply that d has length $3\lambda + 1$, so we pose D_0, D_1, D_2 such that $d = D_0 + \beta^\lambda D_1 + \beta^{2\lambda} D_2$ with D_0 and D_1 of length λ and D_2 of length $\lambda + 1$.

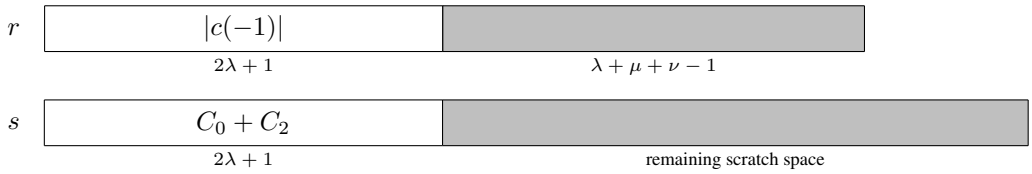
Note that $C_1 + C_3 = C_0 + C_2 - (C_0 - C_1 + C_2 - C_3) = C_0 + C_2 - c(-1)$, so:

$$d = C_0 + C_2 - c(-1) + \beta^\lambda \cdot (C_0 + C_2).$$

The computation of d goes as follows.

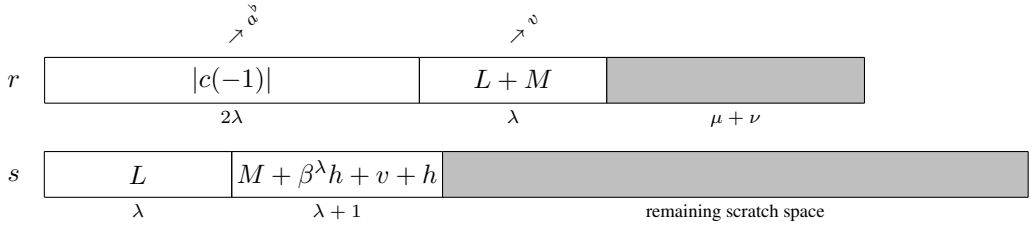


We compute D_0 at s, D_1 at $r + 2\lambda$ and D_2 at $s + \lambda$. The first step is to write $C_0 + C_2$ in s. Noticing that $c(1) + c(-1) = 2 \cdot (C_0 + C_2)$, this is easy to do with one long addition or subtraction (depending on ϵ) of r and s (lines 26-29), and then a logical shift on the result to do the division by 2 (line 30).



Let us note h the highest limb of $C_0 + C_2$ and split the 2λ remaining limbs into two subwords L and M, such that $C_0 + C_2 = L + \beta^\lambda M + \beta^{2\lambda} h$.

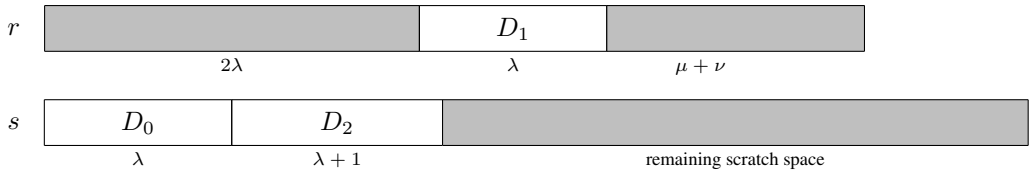
We add L and M together and write the result in $r + 2\lambda, \lambda$, the location of D_1 (line 31 Alg. 13). This overwrites the most significant limb of $|c(-1)|$, but it is still stored in $a^\flat$. The carry v is propagated and added to D_2, or $s + \lambda$, which already contains the higher half of $C_0 + C_2$ (line 32).



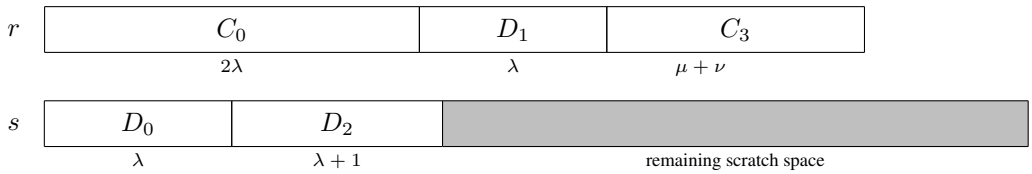
The only thing left to do in the computation of d is to subtract $c(-1)$. The `if` statement at line 33 does so (by adding or subtracting $|c(-1)|$ depending on ϵ). As D_0 and D_1 are not stored contiguously, we have to subtract both halves of $c(-1)$ separately. The ADD_NC and SUB_NC are variants of addition and subtraction that take a carry input to add or subtract to the result, they allow to propagate the carry from the subtraction of the lower half of $c(-1)$ to the upper half. The carry of the upper half subtraction and $a^\flat$ (the high limb of $|c(-1)|$) are propagated to D_2.

At that point, we can check the following:

$$\begin{aligned}
&\texttt{value}(s,\lambda) + \beta^{\lambda}\texttt{value}(r+2\lambda,\lambda) + \beta^{2\lambda}\texttt{value}(s+\lambda,\lambda+1)\\
&= L + \beta^{\lambda}(L + M - \beta^{\lambda}v) + \beta^{2\lambda}(M + \beta^{\lambda}h + v + h) - c(-1)\\
&= L + \beta^{\lambda}M + \beta^{2\lambda}h + \beta^{\lambda}(L + M + \beta^{2\lambda}h) - c(-1)\\
&= C_0 + C_2 + \beta^{\lambda}(C_0 + C_2) - c(-1)\\
&= d.
\end{aligned}$$



5.2.4 *Evaluation in* 0 *and* $+\infty$ *(Alg. 13, lines 41-45).* We recursively compute C_0 in $(r, 2\lambda)$ and C_3 in $(r+3\lambda, \mu+\nu)$. The upper half of s is still available and is used as scratch space. As the operands can be very unbalanced in the case of C_3, we have used the generic multiplication (see Section 5.3) instead of calling Toom-2 or Toom-2.5 directly.



5.2.5 *Recomposition (Alg. 13, lines 46-56).* Let us note L_0 and H_0 the two halves of C_0, and split $C_3 = c(\infty)$ into L_∞ and H_∞ of length λ and $\mu + \nu - \lambda$ respectively.

$$\begin{aligned}
C_0 &= L_0 + \beta^{\lambda}H_0\\
C_3 &= L_\infty + \beta^{\lambda}H_\infty
\end{aligned}$$

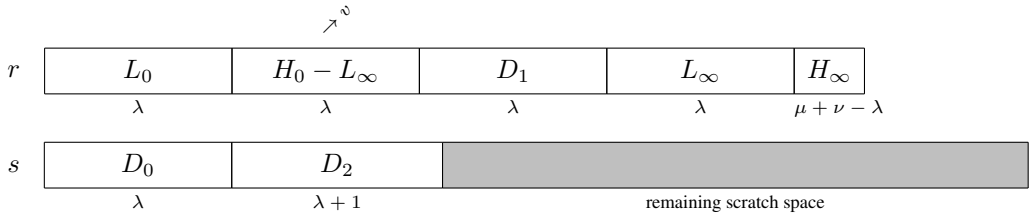
The product $c(\beta^{\lambda}) = a(\beta^{\lambda})b(\beta^{\lambda})$ can be expressed only in terms of C_0, C_3 and d:

$$\begin{aligned}
&C_0 + \beta^{\lambda}d + \beta^{3\lambda}C_3 - C_0\beta^{2\lambda} - \beta^{\lambda}C_3\\
&= C_0 + \beta^{\lambda}(C_1 + C_3) + \beta^{2\lambda}(C_0 + C_2) + \beta^{3\lambda}C_3 - C_0\beta^{2\lambda} - \beta^{\lambda}C_3\\
&= C_0 + \beta^{\lambda}C_1 + \beta^{2\lambda}C_2 + \beta^{3\lambda}C_3\\
&= c(\beta^{\lambda}).
\end{aligned}$$

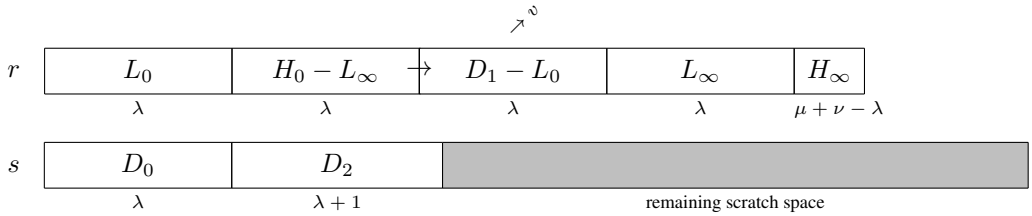
This implies the following decomposition for $c(\beta^{\lambda})$:

$$\begin{aligned}
c(\beta^{\lambda}) &= C_0 + \beta^{\lambda}d + \beta^{3\lambda}C_3 - C_0\beta^{2\lambda} - \beta^{\lambda}C_3\\
&= L_0 + \beta^{\lambda}(D_0 + (H_0 - L_\infty)) + \beta^{2\lambda}(D_1 - L_0 - H_\infty) + \beta^{3\lambda}(D_2 - (H_0 - L_\infty)) + \beta^{4\lambda}H_\infty.
\end{aligned}$$

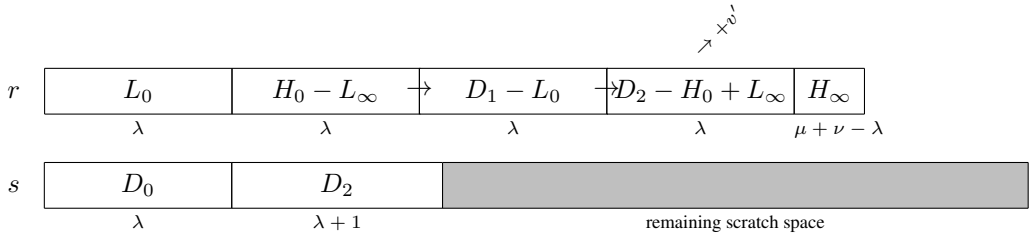
The first step is to subtract L_∞ from H_0 at $r + \lambda$ (lines 46-47). The borrow is stored in v. The variable v' contains the sum of v and the high limb of D_2.



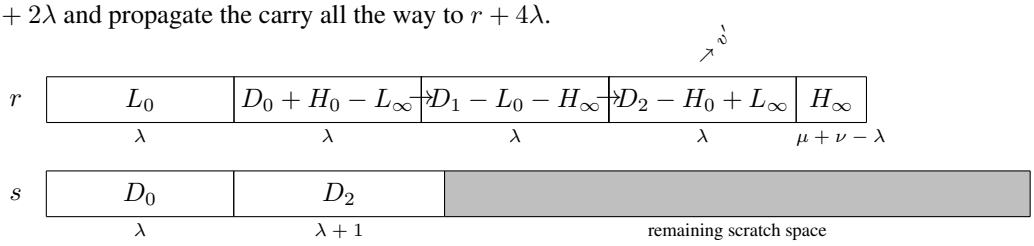
The next step is to subtract L_0 from $r + 2\lambda$ (line 48). The borrom v from the previous computation at $r + \lambda$ is also propagated into $r + 2\lambda$. The final carry is stored in v.



We then subtract $H_0 - L_\infty$ from D_2 at $r + 3\lambda$, propagating the previous carry (line 49). The carry out is accumulated in v'.



We add D_0 at $r + \lambda$, and propagate the carry all the way to $r + 4\lambda$ (as the parameter 3λ is passed in the ADD call at line 50 of Alg. 13). Similarly, we then subtract H_∞ from $r + 2\lambda$ and propagate the carry all the way to $r + 4\lambda$.



We then only have to propagate v' to $r + 4\lambda$ to finish the computation. The proof that this propagation does not overflow is much more straightforward than for Toom-2. Indeed, this time there is only one carry to propagate, so we can rely on the fact that we know for sure that the final result fits in space $3\lambda + \mu + \nu$.

5.3 General case

When the two operands have similar sizes, we can compute their product using either Toom-2 or Toom-2.5. When the operand sizes are very unbalanced, none of these algo-

rithms can be used directly. In this case, we perform a block product by decomposing the larger operand in blocks of sizes $3/2$ times that of the smaller operand, and calling Toom-2.5 repeatedly (Alg. 15).

Algorithm 15 General case multiplication

Require: $\texttt{valid}(r, m+n), \texttt{valid}(a, m), \texttt{valid}(b, n)$
Require: $0 < n \leq m$
Ensure: $\texttt{value}(r, m+n) = \texttt{value}(a, m) \cdot \texttt{value}(b, n)$

1: **function** MUL(r, a, b, m, n)
2: **if** $y < 30$ **then** ▷ Small operands, use schoolbook algorithm.
3: MUL_BASECASE(r, a, b, m, n)
4: **else**
5: $k \leftarrow 64$
6: $s \leftarrow$ ALLOC$(5n + 128)$ ▷ Allocate sufficiently large scratch space.
7: **if** $2m \geq 5n$ **then** ▷ Unbalanced operands, use block product.
8: $m' \leftarrow 3n/2$ ▷ Block size.
9: $w \leftarrow$ ALLOC$(4n)$ ▷ Allocate workspace to store Toom-2.5 result.
10: $u \leftarrow m$ ▷ Remaining section of a to multiply.
11: TOOM32_MUL(r, a, b, s, m', n, k)
12: $u \leftarrow u - m'$
13: $a' \leftarrow a + m'$
14: $r' \leftarrow r + m'$
15: **while** $u \geq 2n$ **do**
16: TOOM32_MUL(w, a', b, s, m', n, k)
17: $v \leftarrow$ ADD_N_IN_PLACE(r', w, n) ▷ Add result to subtotal.
18: COPY$(r + n, w + n, m')$ ▷ Continue the addition.
19: INCR$(r + n, v)$ ▷ Propagate the carry.
20: $u \leftarrow u - m'$
21: $a' \leftarrow a' + m'$
22: $r' \leftarrow r' + m'$
23: **if** $n \leq u$ **then** ▷ Multiply the last block.
24: **if** $4m < 5n$ **then**
25: TOOM22_MUL(w, a', b, s, u, n, k)
26: **else**
27: TOOM32_MUL(w, a', b, s, u, n, k)
28: **else** ▷ Operand sizes are reversed
29: MUL(w, b, a', n, u)
30: $v \leftarrow$ ADD_N_IN_PLACE(r', w, n)
31: COPY$(r + n, w + n, u)$
32: INCR$(r + n, v)$
33: **else**
34: **if** $4m < 5n$ **then**
35: TOOM22_MUL(r, a, b, s, m, n, k)
36: **else**
37: TOOM32_MUL(r, a, b, s, m, n, k)

This function is meant to be exposed to the user, with previous multiplication routines left internal. It calls other multiplication algorithms that depend on the operand sizes. If one operand is very small, the schoolbook multiplication is the best one (line 3). Otherwise, if the operands are of sufficiently similar sizes, then we simply call Toom-2 or Toom-2.5 (line 34 of Alg. 15). Finally, if the operands are large and very unbalanced, we need to perform a block product.

The main loop invariants are as follow:

$$\begin{aligned}\texttt{value}(r, m+n-u) &= \texttt{value}(a, m-u) \cdot \texttt{value}(b, n)\\ a' &= a + m - u\\ r' &= r + m - u.\end{aligned}$$

At the beginning of a loop iteration, a' points to the first limb of a that has not been multiplied yet, and r' points to the zone in r where the next subresult should be added. Note that the first n limbs after r' already contain a part of the subtotal.

After multiplying (a', m') by (b, n), we need to add the product (stored in w) to r'. This is done in lines 17-19. Instead of performing one addition of length $m' + n$, we take advantage of the fact that only the first n limbs of r' are occupied. We perform an addition of length n on that zone and simply copy over the rest of w to $r' + n$, and then propagate the carry of the addition.

What happens after the loop is very similar to one last iteration of the loop. The only change is that the last block of a that is left to multiply has length u, not exactly $3n/2$. Depending on the ratio between n and u, either Toom-2 or Toom-2.5 is called. If $u < n$, the recursive call at line 29 will necessarily jump to line 34 and call one of these two functions, but with the order of the operands reversed to fulfill the precondition that the larger operand is first.

This algorithm is not exactly identical to GMP's. GMP implements a third Toom-Cook algorithm, `toom42`, which splits its larger operand into four parts and the smaller into two. The general case uses this algorithm and splits its large operand in blocks of length $2n$. However, as `toom42` is never called recursively, the asymptotic complexity is similar (the exponent is the same, and the constant is about 7% worse). We did not verify `toom42`, as it did not seem to present many new challenges compared to the other Toom-Cook algorithms and the performance difference is not egregious.

6. SQUARE ROOT

The GMP square root algorithm consists in four functions: two base case algorithms, for operands of length 1 and 2 (`sqrtrem1` and `sqrtrem2` respectively), a general case algorithm (`dc_sqrtrem`) that assumes its operand is normalized, and a wrapper function `sqrtrem` that handles normalization. It computes the square root s and the remainder r of its operand a such that $a = s^2 + r$ and $0 \leq r \leq 2s$, or equivalently, $s^2 \leq a < (s+1)^2$.

The notion of normalization used here is slightly different from the division. The square root algorithms that expect a normalized operand expect its high limb to be greater than or equal to $\beta/4$, rather than $\beta/2$. In other terms, one of its two most significant bits must be set. Note that the square root of such a number is normalized in the sense of the division, which the general case square root algorithm makes use of.

6.1 Square root, $n = 1$

The `sqrtrem1` function computes the square root of a normalized one-limb machine integer. It is used as the base case of GMP's divide-and-conquer square root algorithm. We formally verified its correctness in previous work [21] using Why3 and the Gappa tool [9]. Its implementation relies on C-specific features, such as the way numbers are represented in memory and the semantics of logical shifts for signed and unsigned types. Thus, a pseudocode would not make much sense and would be less readable than the actual C code. For the sake of completeness, we include a slightly edited version of the GMP C implementation in Alg. 16 and outline the algorithm briefly.

Algorithm 16 Square root of a limb.

```
1  #define MAGIC 0x10000000000
2  /* 0xffe7debbfc < MAGIC < 0x232b1850f410 */
3
4  static mp_limb_t mpn_sqrtrem1 (mp_ptr rp, mp_limb_t a0) {
5    mp_limb_t a1, x0, x1, x2, c, t, t1, t2, s;
6    unsigned abits = a0 >> (64 - 1 - 8);
7    x0 = 0x100 | invsqrttab[abits - 0x80];
8    // x0 is the first approximation of 1/sqrt(a0)
9    a1 = a0 >> (64 - 1 - 32);
10   t1 = (mp_limb_signed_t) (0x2000000000000 - 0x30000 - a1 * x0 * x0) >> 16;
11   x1 = (x0 << 16) + ((mp_limb_signed_t) (x0 * t1) >> (16+2));
12   // x1 is the second approximation of 1/sqrt(a0)
13   t2 = x1 * (a0 >> (32-8));
14   t = t2 >> 25;
15   t = ((mp_limb_signed_t) ((a0 << 14) - t * t - MAGIC) >> (32-8));
16   x2 = t2 + ((mp_limb_signed_t) (x1 * t) >> 15);
17   c = x2 >> 32;
18   // c is a full limb approximation of sqrt(a0)
19   s = c * c;
20   if (s + 2*c <= a0 - 1) {
21      s += 2*c + 1;
22      c++;
23    }
24   *rp = a0 - s;
25   return c;
26 }
```

The program is best understood as a fixed-point arithmetic algorithm that computes the square root of a real number $a \in [0.25, 1]$ using Newton's method. Assuming we want to compute the square root of the integer $a_0 \in [2^{62}, 2^{64} - 1]$, we define $a = 2^{-64}a_0$. We start by fetching an 8-bit approximation x_0 of $a^{-1/2}$ from a precomputed table (line 7). Let us pose ε_0 such that $x_0 = a^{-1/2}(1 + \varepsilon_0)$. We have $|\varepsilon_0| \leq 2^{-8}$.

The algorithm consists in performing two steps of Newton's method to reach a 32-bit approximation of $a^{-1/2}$. Equivalently, we are looking for roots of the function $f(x) = x^{-2} - a$. Given $x_i = a^{-1/2}(1+\varepsilon_i)$, we compute $x_{i+1} = x_i - f(x_i)/f'(x_i) = x_i(3 - ax_i^2)$. The relative error ε_i decreases quadratically. More precisely, we find $|\varepsilon_{i+1}| \approx \frac{3}{2} \cdot |\varepsilon_i^2|$. Intuitively, we need about two steps to reach a precision of 32 bits. The reason why we

are computing $a^{-1/2}$ rather than $a^{1/2}$ is that the Newton iteration for $a^{1/2}$ would require a division at each step, so it would be much more costly than the one for $a^{-1/2}$.

The algorithm computes x_1 between lines 9 and 11, and then x_2 between lines 13 and 17. As part of the computation of x_2, there is an extra multiplication by a in order to compute the square root of a rather than its inverse. We also multiply x_2 by 2^{32} to get back to computing the square root of a_0 rather than a. At line 17, we obtain $c = \sqrt{a_0}(1 + \varepsilon_2)$.

At this point, we have $\varepsilon_2 \in [-2^{-32}, 0]$. In other terms, either c is the square root of a_0, or it is too small by one. The fact that $|\varepsilon_2|$ is smaller than 2^{-32} is far from obvious. Indeed, the constant $3/2$ in the growth of ε_i is problematic. Furthermore, we are computing using fixed-precision arithmetic rather than real numbers, so we introduce rounding errors. However, the initial approximation x_0 is actually a bit tighter than 8 bits ($|\varepsilon_0|$ is smaller than about $2^{-8.5}$ for all values of a_0), which offsets the precision loss from the constant $3/2$. Furthermore, the computations of x_1 and x_2 involve magic constants that would not be present in the theoretical steps of the Newton iteration (`0x30000` at line 10 and `MAGIC` at line 15). These extra constants offset the rounding errors and ensure that the final approximation is always by default.

The final step of the algorithm (lines 19-25) performs a final adjustment step (in case we computed $\sqrt{a_0} - 1$ rather than $\sqrt{a_0}$) and computes the remainder.

6.2 Square root, $n = 2$

The `sqrtrem2` function (Alg. 17) computes the square root of a, which must be two limbs long. It takes two destinations operands, s to store the square root of a and r to store the remainder. It returns the high limb c of the remainder, such that $\texttt{value}(a, 2) = s[0]^2 + \beta c + r[0]$. The parameters r and s are pointers that only need space for 1 limb. We assume a to be normalized such that its high limb is greater than or equal to $\beta/4$. The algorithm computes an initial estimate of the square root by calling `sqrtrem1` on the high limb of a, and then adjusts toward the correct square root.

6.2.1 *Initial estimate.* After the call to `sqrtrem1` at line 2, we have $s_0^2 + r[0] = a_1$ and $a = 2^{64}a_1 + a_0$. The value $2^{32}s_0$ is the initial estimate for the square root of a. We compute an initial remainder r_0 such that $2^{33}r_0 + (a[0] \bmod 2^{33}) = \beta r[0] + a[0]$.

6.2.2 *Adjustment.* We compute the quotient q of r_0 by s_0, such that at line 6, $s_0 q$ is somewhat close to r_0 and $q < 2^{32}$. We pose $s_1 = 2^{32}s_0 + q$ as a candidate square root. Indeed, $2^{33}qs_0 \approx \beta r[0] + a[0]$, so at line 8: $s_1^2 = \beta s_0^2 + 2^{33}qs_0 + q^2 \approx \beta a[1] + a[0] + q^2$. More precisely,

$$s_1^2 = \beta a[1] + a[0] + q^2 - (2^{33}(r_0 - qs_0) + a[0] \bmod 2^{33}).$$

We pose $u = r_0 - qs_0$, and r_1, c such that $r_1 + \beta c = 2^{33}(r_0 - qs_0) + a[0] \bmod 2^{33}$, $r_1 \in [0, \beta - 1]$. Therefore, at line 11 we have $s_1^2 + r_1 + \beta c = a + q^2$. At lines 12-13, we subtract q^2 from $r_1 + \beta c$ in a way that avoids arithmetic underflows. At that point, $s_1^2 + r_1 + \beta c = a$.

6.2.3 *No underapproximation of the square root.* Let us show that s_1 is not an underapproximation of the square root, or in other terms, $\lfloor \sqrt{a} \rfloor \leq s_1$. This is equivalent to $a < (s_1 + 1)^2$, or $r_1 + \beta c \leq 2s_1$. We have $r_1 + \beta c = 2^{33}u + a_l - q^2$.

If $q_0 < 2^{32}$, then $q = q_0$ and $u = r_0 \bmod s_0 < s_0$, and $a_l < 2^{33}$, so $r_1 + \beta c \leq 2^{33}s_0 \leq 2s_1$.

The only remaining case is $q_0 = 2^{32}, q = 2^{32}-1$. In this case, we notice that $r[0] \leq 2s_0$ (postcondition of `sqrtrem1`). This implies $r_0 \leq 2^{32}s_0 + a_h$, so $u = r_0 - (2^{32}-1)s_0 \leq s_0 + a_h$. We also have $\beta = (q+1)^2$, so:

$$\begin{aligned} 2^{33}u + a_l - q^2 &\leq 2^{33}s_0 + a[0] - q^2 \\ &\leq 2^{33}s_0 + \beta - 1 - q^2 \\ &= 2^{33}s_0 + 2q \\ &\leq 2s_1. \end{aligned}$$

6.2.4 *Avoiding overapproximation.* Let us now show that $s_1 < \lfloor\sqrt{a}\rfloor + 1$, or equivalently, $s_1^2 \leq a$ (no overapproximation of the square root). If we had $s_1^2 > a$, it would imply $r_1 + \beta c < 0$, or equivalently, $c \leq -1$. The conditional at line 14 takes care of this case by adding $2s_1 - 1$ to $r_1 + \beta c$ and removing 1 from s_1, leaving the sum $s_1^2 + r_1 + \beta c$ unchanged and avoiding arithmetic overflows.

Algorithm 17 Square root of a 2-limb number

Require: $\texttt{valid}(a, 2), \texttt{valid}(s, 1), \texttt{valid}(r, 1)$
Require: $a[1] \geq \beta/4$
Ensure: $\texttt{value}(a, 2) = s[0]^2 + \beta \times \texttt{result} + r[0]$
Ensure: $0 \leq \texttt{result} \leq 1$
Ensure: $r[0] + \beta \times \texttt{result} \leq 2s[0]$

1: **function** SQRTREM2(s, r, a)
2: $s_0 \leftarrow$ SQRTREM1$(r, a[1])$
3: $a_h \leftarrow a[0] \gg 33$
4: $r_0 \leftarrow (r[0] \ll 31) + a_h$
5: $q_0 \leftarrow \lfloor r_0/s_0 \rfloor$ ▷ $q_0 \leq 2^{32}$.
6: $q \leftarrow q_0 - (q_0 \gg 32)$ ▷ If $q_0 = 2^{32}$, reduce it by 1.
7: $u \leftarrow r_0 - qs_0$
8: $s_1 \leftarrow (s_0 \ll 32) + q$
9: $c \leftarrow u \gg 31$
10: $a_l \leftarrow a[0] \bmod 2^{33}$
11: $r_1 \leftarrow (u \ll 33) + a_l$
12: $c \leftarrow c - (r_1 < q^2)$ ▷ $-1 \leq c$
13: $r_1 \leftarrow r_1 - q^2 \bmod \beta$
14: **if** $c < 0$ **then** ▷ Square root too large, adjust.
15: $r_1 \leftarrow r_1 + s_1 \bmod \beta$
16: $c \leftarrow c + (r_1 < s_1)$ ▷ Carry propagation.
17: $s_1 \leftarrow s_1 - 1$
18: $r_1 \leftarrow r_1 + s_1 \bmod \beta$
19: $c \leftarrow c + (r_1 < s_1)$ ▷ Carry propagation.
20: $r[0] \leftarrow r_1$
21: $s[0] \leftarrow s_1$
22: **return** c

6.3 Square root, general case

Our Why3 proof of the general case divide-and-conquer square root algorithm is largely lifted from Bertot et al. [3], with only minor adjustments to account for small changes in the GMP implementation since the publication of their article. For the sake of completeness, the signature and specification of the algorithm are given in Alg. 18.

Algorithm 18 Square root of a normalized integer (specification)

Require: $\texttt{valid}(a, 2n)$, $\texttt{valid}(s, n)$, $\texttt{valid}(w, \lfloor n/2 \rfloor + 1)$
Require: $1 \leq n$
Require: $a[2n-1] \geq \beta/4$
Ensure: $\texttt{value}(s,n)^2 + \texttt{value}(a,n) + \beta^n \texttt{result} = \texttt{value}(\texttt{old } a, 2n)$
Ensure: $\texttt{value}(a,n) + \beta^n \texttt{result} \leq 2 \times \texttt{value}(s,n)$
Ensure: $0 \leq \texttt{result} \leq 1$
function DC_SQRTREM(s, a, w, n)

6.4 Square root, normalizing wrapper

The previous "general case" algorithm still requires its input to have even length and to be normalized (the highest limb of a must be greater than or equal to $\beta/4$). The final algorithm (Alg. 19) is essentially a wrapper around the two previous ones that normalizes its operand, calls the appropriate square root function, and denormalizes the result. It returns the size of the remainder in limbs. If the result is 0, the operand is a perfect square.

The key idea of this algorithm is the following lemma from Bertot et al. [3]:

LEMMA NORMALIZATION. *Let N, N_1, S_1, c such that $S_1^2 \leq N_1 < (S_1+1)^2$, $N_1 = 2^{2c}N$. Let S, s_0 such that $S_1 = 2^c S + s_0$, $0 \leq s_0 < 2^c$. Then $S^2 \leq N < (S+1)^2$.*

This justifies that simply denormalizing the square root of the normalized operand gives the correct square root.

We first compute c, the floor of half the number of leading zeros of the high limb of a. This means that shifting a to the left by $2c$ will multiply it by a power of 2 such that at most one leading zero remains, which is exactly the precondition of the previous square root algorithms.

6.4.1 *Special case $n = 1$.* This case is a direct application of the lemma above. If the operand is not already normalized, we shift it to the left by $2c$ and shift the result to the right by c. The lemma ensures that this yields the correct square root, and we compute a remainder straightforwardly.

6.4.2 *General case.* We compute $k = \lfloor (n+1)/2 \rfloor$ such that $2k = n + (n \bmod 2)$. If the conditional at line 19 is true, then a is either not normalized or of odd length, so we must normalize it before calling `dc_sqrtrem`. After line 27, this is done and we have $\texttt{value}(t, 2k) = 2^{2c}\texttt{value}(a, n)$. The value of c is incremented by 32 if n is odd, so that we shift a by an extra limb (which is what the computation of t' does) and end up with an even-length number.

At that point, we can call `dc_sqrtrem`. The normalization lemma implies that $\lfloor \frac{\texttt{value}(s,k)}{2^c} \rfloor$ is the correct square root (using the same variables as the lemma, $S = \lfloor S_1 2^{-c} \rfloor$). We still need to compute the remainder.

Algorithm 19 Square root of an integer

Require: $\texttt{valid}(s, \lfloor n/2 \rfloor + 1), \texttt{valid}(r, n), \texttt{valid}(a, n)$
Require: $1 \leq n$
Require: $a[n-1] > 0$
Ensure: $\texttt{value}(a, n) = \texttt{value}(s, \lfloor n/2 \rfloor + 1)^2 + \texttt{value}(r, \texttt{result})$
Ensure: $\texttt{value}(r, \texttt{result}) \leq 2 \times \texttt{value}(s, n)$
Ensure: $\texttt{result} > 0 \Rightarrow r[\texttt{result} - 1] > 0$

1: **function** SQRTREM(s, r, a, n)
2: $h \leftarrow a[n-1]$
3: $c = \text{COUNT\_LEADING\_ZEROS}(h)/2$
4: **if** $n = 1$ **then**
5: **if** $c = 0$ **then**
6: $s[0] = \text{SQRT1}(r, h)$
7: **else**
8: $h' \leftarrow h \ll 2c$
9: $s' \leftarrow \text{SQRT1}(r, h') \gg c$
10: $s[0] \leftarrow s'$
11: $r[0] \leftarrow h - s'^2$
12: **if** $r[0] = 0$ **then**
13: **return** 0
14: **else**
15: **return** 1
16: $k \leftarrow (n+1)/2$
17: $w \leftarrow \text{ALLOC}(k/2 + 1)$
18: $t \leftarrow \text{ALLOC}(2k)$
19: **if** $n = 1 \bmod 2 \vee c \neq 0$ **then**
20: $t[0] \leftarrow 0$
21: $t' \leftarrow t + (n \bmod 2)$
22: **if** $c \neq 0$ **then**
23: $\text{LSHIFT}(t', a, n, 2c)$
24: **else**
25: $\text{COPY}(t', a, n)$
26: **if** $n \bmod 2 = 1$ **then**
27: $c \leftarrow c + 32$
28: $r_l \leftarrow \text{DC\_SQRTREM}(s, t, w, k)$
29: $s_0 \leftarrow \text{ALLOC}(1)$
30: $s_0[0] \leftarrow s[0] \bmod 2^c$
31: $r_l \leftarrow r_l + \text{ADDMUL\_1}(t, s, k, 2s_0[0])$
32: $b \leftarrow \text{SUBMUL\_1}(t, s_0, s_0[0], 1)$
33: **if** $k > 1$ **then**
34: $b \leftarrow \text{SUB\_1\_IN\_PLACE}(t + 1, b, k - 1)$
35: $r_l \leftarrow r_l - b$
36: $\text{RSHIFT\_IN\_PLACE}(s, k, c)$
37: $t[k] \leftarrow r_l$

38: $c_2 \leftarrow 2c$
39: **if** $c_2 < 64$ **then**
40: $k \leftarrow k + 1$
41: **else**
42: $t \leftarrow t + 1$
43: $c_2 \leftarrow c_2 - 64$
44: **if** $c_2 \neq 0$ **then**
45: RSHIFT(r, t, k, c_2)
46: **else**
47: COPY(r, t, k)
48: $r_n \leftarrow k$
49: **else**
50: COPY(r, a, n)
51: $h \leftarrow$ DC_SQRTREM(s, r, w, k) ▷ $0 \leq h \leq 1$
52: $r[k] \leftarrow h$
53: $r_n \leftarrow k + h$
54: **while** $r[r_n - 1] = 0$ **do**
55: $r_n \leftarrow r_n - 1$
56: **if** $r_n = 0$ **then**
57: **break**
58: **return** r_n

At line 29, we pose $S_1 = \texttt{value}(s, k)$, $R_1 = \texttt{value}(t, k) + \beta^k r_l$, $N = \texttt{value}(a, n) =$, $N_1 = 2^c N$. We have $S_1^2 + R_1 = N_1$. We pose $s_0 = S_1 \bmod 2^c$. N_1 can be written as $(S_1 - s_0)^2 + 2S_1 s_0 - s_0^2 + R_1$. We add $2S_1 s_0$ to t at line 31, and then subtract s_0^2 at line 32. After line 32, $\texttt{value}(t, k) + \beta^k r_l - \beta b = R_1 + 2S_1 s_0 - s_0^2$. Note that the borrow b has not yet been propagated, instead the subtraction was only over a length of 1. The propagation occurs at lines 33-34, and after line 35 we have $\texttt{value}(t, k) + \beta^k r_l = R_1 + 2S_1 s_0 - s_0^2$. At line 37 we write r_l at $t[k]$ such that $\texttt{value}(t, k + 1) = R_1 + 2S_1 s_0 - s_0^2$.

We pose S such that $S_1 = s_0 + 2^c S$. The normalization lemma implies that S is the square root of N. At line 36, we denormalize s such that $\texttt{value}(s, k) = S$. Furthermore, $S_1 - s_0 = 2^c S$, so t holds the remainder of the square root: $\texttt{value}(t, k + 1) = R_1 + 2S_1 s_0 - s_0^2 = N_1 - 2^{2c} S^2 = 2^{2c}(N - S^2)$. The only remaining thing to do is to shift t by $2c$. As c can be up to 63, $2c$ can be up to 126 and `rshift` only accepts parameters smaller than 64. The conditional at lines 39-43 takes care of this. If $c_2 \geq 64$, then the low limb of t is all zeroes, so we can cheaply shift t by 64 by simply incrementing the pointer t. At line 44, $\texttt{value}(t, k) = 2^{c_2}(N - S^2)$ and $0 \leq c_2 \leq 63$, so we only have to shift t to the right to get the correct remainder.

6.4.3 *Normalized case.* If the operand is already normalized and of even length, we can simply call `dc_sqrtrem`. This is done at lines 50-53.

6.4.4 *Normalizing the remainder.* At line 54, r_n contains an upper bound on the length of the remainder. However, any number of high limbs may be zero. We simply iterate to return r_n such that $r_n = 0$ or $r[r_n - 1] > 0$.

7. RELATED WORK

In this work, we have used the Why3 tool [4, 13, 11] to formally verify GMP's integer arithmetic layer. We obtain a verified and efficient C library. Previous work generally does not deal with a large number of highly optimized algorithms. As far as we know, this work is the first formal verification of a comprehensive arbitrary-precision integer library with comparable performance to the state of the art. Let us now discuss a few examples of existing verifications of multiprecision arithmetic functions or libraries.

Bertot et al. verified GMP's square-root general case algorithm [3] using Coq and the Correctness tool, which translates an imperative program and its specifications into verification conditions to be proved with Coq. Our Why3 proof of the same algorithm is directly inspired from their article. They specify the memory as one large array of machine integers, so their specifications must include additional clauses to tell which memory zones are left unchanged. Their formalization is otherwise quite similar to ours. However, their proof is about 13,000 lines long, which is about as long as all our proofs combined. Indeed, Why3 proofs are partially automatic, while Coq proofs are entirely interactive, so it is not surprising that we enjoy a lower proof effort.

Myreen and Curello verified an implementation of arbitrary-precision integer arithmetic using the HOL4 theorem prover and separation logic [23]. Their library covers the four basic arithmetic operations, but not the square root. They do not attempt to produce highly efficient code. As a result, the algorithms they proved are simpler and less efficient than the optimized ones that we proved. For example, their multiplication algorithm is the schoolbook one. However, their verification goes all the way down to x86 machine code, using formally verified proof-generating compilers and decompilers to do part of the proof on a higher-level implementation. Using these tools, they also manage to avoid most proofs involving pointer reasoning. Their proof effort per algorithm is similar to ours despite them using an interactive tool.

Affeldt used Coq to verify a binary extended GCD algorithm implemented in a variant of MIPS assembly [1], as well as the functions it depends on, such as addition, subtraction and halving. The work encompasses both signed and unsigned integer arithmetic. It uses GMP's number representation and a memory model based on separation logic. The author verifies the algorithm in a pseudo-code language and proves a forward simulation relation between the pseudo-code and the MIPS assembly code to prove the latter's correctness. It makes some simplifying assumptions, such as requiring that the operands share the same length. The GCD algorithm that is verified is not trivial, but it is much less involved and efficient than the one implemented in GMP (which we have not verified). The proof effort is hard to quantify, as the development relies on preexisting frameworks by the same author for pseudocode and assembly code, but it is rather high. The proofs of the algorithms amount to about 15,000 lines of Coq.

Fischer designed a modular exponentiation library [14] verified using Isabelle/HOL and a framework for verifying imperative programs developed by Schirmer [25]. The verified algorithms include multiplication, division and square-and-multiply modular exponentiation. The library is not meant to be as efficient as GMP, as it represents arbitrary-precision integers as garbage-collected doubly-linked lists of machine integers. The algorithms are implemented in a restricted variant of the C language and are automatically transcribed into Isabelle. The functional correctness of the algorithms and the pointer-level correctness of the data structure are proved, but not termination or the absence of arithmetic overflows.

The author reports running into slowdown and memory issues inside the tool due to the great number of invariants and conditions present in the logical context to keep track of aliasing. By contrast, Why3 automatically keeps track of aliases inside its region-based type system, rather than in the logic. This means that the user does not need to mention in specifications and proofs that such and such pointers are not aliased, which would otherwise cause large slowdown issues similar to those reported by Fischer.

Berghofer developed a verified bignum library programmed in the SPARK fragment of the Ada programming language, using a verification framework that sends goals to Isabelle/HOL [2]. The library provides modular exponentiation, as well as the primitives required to implement it: modular multiplication and squaring, modular inverse, and basic operations such as subtraction and doubling. A simple square-and-multiply algorithm is used for modular exponentiation, without the sliding window optimization that GMP and other state-of-the-art libraries implement. The author reports a 150% slowdown compared to OpenSSL for their implementation of RSA using their library. However, OpenSSL uses hand-written assembly code, which accounts for a large part of the discrepancy. The proof effort for the library is only about 2,000 lines of Isabelle written over three weeks, which is surprisingly low, even taking into account the low amount of verified algorithms.

Unlike our semi-automatic Why3 proofs, most of the approaches described above use interactive proof assistants. There have also been efforts to prove arithmetic libraries using automated tools. Schoolderman used Why3 to verify hand-optimised Karatsuba multiplication branch-free assembly routines for the AVR microcontroller architecture [26]. The algorithms are not arbitrary-precision, instead there are many routines, each suited for a particular operand size up to 96×96 bits. The fact that the size of the operands is fixed and relatively small means the loops can be unrolled, which is why the algorithms are branch-free. The size being known also makes the proof much easier for SMT solvers, and the authors only needed to add a very small number of annotations to make the automatic proof succeed.

Finally, Zinzindohoué et al. developed an elliptic curve cryptography library written and verified in F^* that can be extracted to C [29]. It implements the full NaCl API. The integers do not have an arbitrary large size, instead they have a small, fixed size that depends on the choice of elliptic curve. A peculiarity is that only part of a machine word is used for each limb. For instance, the most used curve uses 255-bit integers, which are not stored using 4 limbs of 64 bits, but 5 limbs of 51 bits. This means that arithmetic operations on limbs do not overflow and carry bits do not have to be propagated. The performance of their extracted C code is comparable to state-of-the-art C implementations. Part of the code is now in use in several products such as the Mozilla Firefox web browser.

8. CONCLUSION

We have used the Why3 platform to verify a library that implements state-of-the-art algorithms from the `mpn` layer of GMP for comparison, addition, subtraction, multiplication, division and square root. We have tackled two different algorithms for multiplication, one suited for relatively small numbers (fewer than 1500 bits) and one suited for larger numbers. This verification effort covers functional correctnes and implementation aspects such as memory safety and absence of arithmetic overflows.

Due to the partially automatic nature of Why3 proofs, we are able to perform such a verification work with a high but reasonable amount of proof effort. The Why3 development is about 13000 lines long, split between about 4500 lines of program code and 8500 lines

of annotations and proofs. The extracted library consists of about 2500 lines of C code. The proof-to-code ratio is worse than for most existing Why3 proofs. Indeed, non-linear arithmetic is difficult for automated provers, and the fact that the operands have unknown lengths prevents the use of bit-blasting techniques. To make up for this, we had to write a large amount of assertions that are practically transcriptions of paper proofs. As such, for the mathematical parts of the proofs, the way we used Why3 was not much faster than the way we would have used an interactive tool such as Coq.

While the mathematical parts of the proof still require a lot of user guidance, most subgoals related to memory safety and arithmetic overflows can be discharged by automated theorem provers. Our C memory model and Why3's static treatment of aliasing make the memory management part come almost for free. The GMP source code is rather well documented, which made the specifications and invariants easy to write except for some internal functions. All in all, the total proof effort is lower than for most related works which use interactive proof assistants.

Using Why3's extraction mechanism, we obtain a verified C library that is compatible with GMP. Our implementation contains all the algorithmic tricks that can be found in the corresponding GMP functions, which makes the extracted code performance-competitive with the non-assembly version of GMP [24]. What we have verified is not strictly GMP's implementation, but rather their WhyML transcription and, by extraction, a new C library that uses the same algorithms. However, the fact that our implementation closely mirrors GMP's C code does give an additional measure of trust in GMP's own correctness. While we have not found any actual bug in GMP, we found something that can be seen as a bug in the proof of GMP's Toom-2 algorithm. Indeed, although it turned out that it was correct, the complexity of our proof led the GMP developers to modify the algorithm to match the correctness proof that they expected.

Our library does not implement GMP's full interface. We intend to verify the missing primitives, e.g. side-channel resistant modular exponentiation and base conversions, in future work. Another natural extension would be to tackle the `mpz` layer of GMP, which is the higher-level interface for signed arbitrary-precision arithmetic. It is essentially a wrapper around the `mpn` layer that we tackled in this work. It contains relatively little arithmetic but poses many interesting verification challenges in its memory management.

Acknowledgements

We gratefully thank Guillaume Melquiond and the anonymous reviewers for their extensive feedback on earlier versions of this article.

References

[1] Reynald Affeldt. On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering*, 9(2):59–77, 2013.

[2] Stefan Berghofer. Verification of dependable software using SPARK and Isabelle. In Jörg Brauer, Marco Roveri, and Hendrik Tews, editors, *6th International Workshop on Systems Software Verification*, volume 24 of *OpenAccess Series in Informatics (OASIcs)*, pages 15–31, Dagstuhl, Germany, 2012.

[3] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. `https://hal.inria.fr/hal-00790310`.

[5] Marco Bodrato and Alberto Zanoni. Integer and polynomial multiplication: Towards optimal Toom-Cook matrices. In *2007 International Symposium on Symbolic and Algebraic Computation*, pages 17–24. ACM, 2007.

[6] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[7] Martin Clochard. Preuves taillées en biseau. In *vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA)*, Gourette, France, January 2017.

[8] Stephen A. Cook. *On the minimum computation time of functions*. PhD thesis, Department of Mathematics, Harvard University, 1966.

[9] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1):1–20, 2010.

[10] Edsger W. Dijkstra. *A discipline of programming*, volume 1. Prentice-Hall Englewood Cliffs, 1976.

[11] Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013.

[12] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.

[13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128, Heidelberg, Germany, March 2013.

[14] Sabine Fischer. Formal verification of a big integer library. In *DATE Workshop on Dependable Software Systems*, 2008.

[15] Robert W. Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[16] Léon Gondelman. *A Pragmatic Type System for Deductive Software Verification*. PhD thesis, Université Paris-Saclay, December 2016.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[18] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet Physics Doklady*, volume 7, pages 595–596, 1963.

[19] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[20] Guillaume Melquiond and Raphaël Rieu-Helft. A Why3 framework for reflection proofs and its application to GMP's algorithms. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, pages 178–193, Oxford, United Kingdom, July 2018.

[21] Guillaume Melquiond and Raphaël Rieu-Helft. Formal verification of a state-of-the-art integer square root. In *IEEE 26th Symposium on Computer Arithmetic (ARITH)*, Kyoto, Japan, June 2019.

[22] Niels Moller and Torbjörn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60:165–175, 2011.

[23] Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In Georges Gonthier and Michael Norrish, editors, *3rd International Conference on Certified Programs and Proofs (CPP)*, volume 8307 of *Lecture Notes in Computer Science*, pages 66–81, Melbourne, Australia, December 2013.

[24] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. How to get an efficient yet verified arbitrary-precision integer library. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, pages 84–101, Heidelberg, Germany, July 2017.

[25] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 398–414, 2005.

[26] Marc Schoolderman. Verifying branch-free assembly code in Why3. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 66–83, 2017.

[27] Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.

[28] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

[29] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. Cryptology ePrint Archive, Report 2017/536, 2017. `https://eprint.iacr.org/2017/536`.